CS145 Kaggle Leaf Project 3A

Alex Hesselgrave (904273474),
Angela Lin (604268983),
Minghong Zhou (004424670),
Linxi Han(104588802),
Lingzhi Li(504589866)

Abstract: In this report we examine the accuracy and quality of multiple classifiers and ensembles to classify leaves based on physical qualities. Our data was provided by Kaggle's leaf classification challenge, and our accuracy was determined both by our code in Python library scikit-learn and Kaggle's scoring algorithm. Each team member chose a unique ensemble that has not been used on Kaggle yet, each with an explanation of the ensemble algorithm and analysis of the accuracy for better or for worse. We also provide a link to our Github with instructions to run each ensemble mentioned in the report.

Introduction

This report will analyze several different classifiers and ensembles as well as their effectiveness in classifying leaf species given by the Kaggle leaf classification challenge. We will begin by answering some preliminary questions as well as analyzing decision tree and random forest classifiers as per the spec. Finally, we will each run and analyze different ensembles and compare the results to each other to find the best qualifier and hyperparameters for the challenge.

We would also like to define a few terms that are used throughout the paper. **Accuracy** is defined as the number of correctly classified leaf species versus the total number of test rows when testing a classifier. **Log loss** is another metric that is formulaically given by

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij}),$$

"where N is the number of samples or instances, M is the number of possible labels, y_{ij} is a binary indicator of whether or not label j is the correct classification for instance i, and p_{ij} is the model probability of assigning label j to instance i" (Collier). As shown in the formula, the classifier has to assign a probability to each class instead of attempting to fit the best class. Then, log loss increases and "punishes" a classifier that shows certainty for a wrong class. A perfect classifier has a log loss of zero, and this is the metric that Kaggle uses to grade submissions.

Finally, this report does discuss in detail about the Python implementation of our ensembles. All our code is provided in the following Github repository. Provided in the repository is a requirements.txt dependency file generated from the Python package manager 'pip'. To run the code, create a virtual environment using virtualenv, source into the environment, and execute "pip install -U -r requirements.txt" in the shell of your choice. After the installation/updating finishes, you will be able to execute any classifier .py file using Python 2.7. The files are appropriately named for the classifier it will train and execute.

Preliminary Questions and Experimentation

Before we begin our analysis, we want to address some preliminary questions about the project and the data. All training data in this project is derived from images of leaves, but

numerically classified in 192 features in a CSV file. Suppose that there is some missing or corrupt data in the training or test data. Our course of action is to omit the row(s) that are corrupt from the appropriate set. The downside is that the more data we omit, the less accurate our classifier will be.

Consider the following scenario. Alice wants to do the following: she wants to train the classifier using the training set and test the classifier using the test set. Based on the output of the test set, she wants to make modifications to her classifier to increase the classifiers accuracy (Question 2b). This idea is not a bad idea because testing the classifier is effectively running the same routine as training, but quantifying the output accuracy. However, if Alice wanted to test the classifier again, she would want to have a new test dataset to avoid any bias of training off the test set.

Finally, let us examine the results of using decision trees versus random forests on the leaf data. Decision tree learning generates a hierarchical structure of tests to reach a final classification, effectively building a flowchart. The tree splits on each feature in an optimal order, usually based on the information gain of each features. However, a deep decision tree tends to overfit and develop high variance of classification. To remedy this, random forests were created. The random forest is an ensemble that consists of a set of decision trees. Random forests use a modified form of bootstrap aggregation, or bagging, to help with the overfitting problem. In typical bagging on a training set of size N, the bagging algorithm will generate *m* subsets by sampling N with replacement, where *m* is the number of trees. Each tree trains itself on its respective subset, and classifications are made by majority vote amongst the trees. Random forest uses a method called **feature bagging** that randomly selects a subset of features at each split. The random factor combined with the voting ensemble helps to keep the variance down.

Now that we understand the high level result of the algorithms, we can show our results. Based on the previous paragraph, we can safely predict that the RandomForestClassifier will be more accurate with less log loss than the DecisionTreeClassifier, and the results support this claim. On average, the DecisionTreeClassifier gave us around 64% accuracy with a log loss of 12.3 while the RandomForestClassifier with 100 trees gave us an accuracy of 98% with a log loss around 0.7. While the number of trees seems high, note that increasing the amount of trees has quickly diminishing returns, sacrificing a lot of computational power for little to no gain in accuracy. For our systems, we empirically determined that 100 trees was a good balance of speed and accuracy.

Finding the Best Ensemble

From this point on, each group member chose an ensemble to execute and analyze for performance in the search for the best ensemble for this problem. We each describe our motivation for choosing the ensemble, the method on how we approach the problem, and the results we get.

Extremely Random Forests (Alex Hesselgrave)

Extremely random forests, also called ExtraTrees, are a further iteration of random forests, which were described in the preliminary analysis. Random forests expanded on decision trees through random feature bagging on each split. ExtraTrees takes the randomness a step forward by randomly picking the feature to split on from the bagging subset rather than choosing the optimal feature based on information gain. I chose this ensemble because I was very impressed with the results of the RandomForestClassifier with minimal tuning, and I was curious to see how an added layer of randomness would compete against an already very accurate classifier.

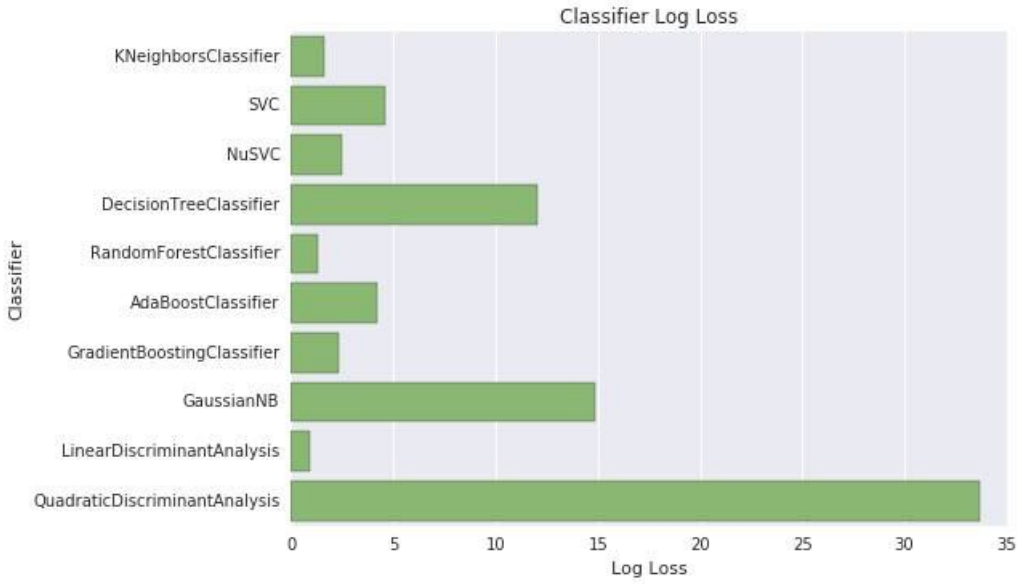
In scikit-learn, the classifiers and ensembles are polymorphic and share the same methods for training and testing, but with different implementations. This made a direct comparison easy to build because I was able to use the same Pandas DataFrame on both classifiers with near identical code. The actual implementation can be seen in extratrees.py in the Github repository.

The results were initially neck-and-neck. Both classifiers were both tuned to have 100 trees with the rest of the parameters being set to default, and there was an incredibly small difference in log loss and accuracy. I began to empirically tune the ExtraTreesClassifier by changing parameters in the constructor. I finalized on the classifier having n_estimators=100, max_features=None, and min_samples_split=1. N_estimators is the number of trees in the ensemble. max_features=None means that the classifier considers all *n* features when deciding where to split instead of the default *sqrt(n)*. This resulted in a small, but noticeable increase in computation time but a slightly lower log loss. Finally, min_samples_split=1 is changed from the default of 2 to split more frequently. Given the amount of trees and the randomness of the ensemble, this immensely improved log loss from around 0.7 to around 0.45, a 35% improvement on the log loss. Note that this log loss was calculated using the method included in scikit-learn. My submission on Kaggle Leaf was given a score of 0.68557, possibly based on a combination of rounding errors in the submission CSV and different implementation of the log loss formula for Kaggle, which as of this writing is rank 597/816.

Although the accuracy and log loss was acceptable, scores on Kaggle indicate it can be better. For the future, I would like to be able to fine tune the parameters even further, possibly firing up a powerful server to handle thousands of trees to consult on. Furthermore, the current dataset is only around 1000 entries, and the classifier can be trained to be much more accurate with more trees trained on tens of thousands of entries.

Voting Classifier - Weighted Average Probabilities (Angela Jin)

Motivation: Based upon preliminary analysis on various classifiers, utilizing existing scikit learn library, we discover that the K-neighbours Classifier, Linear Discriminant Analysis, and Random Forest Classifiers are the best performers. As we can see from the following image, these three classifiers have the lowest log loss when testing against the training data. In order to achieve an even lower log loss, we looked for a way to assemble these three top-performing classifiers.



Method: We used voting classifier to combine these conceptually different machine learning classifiers, and used weighted average probabilities to predict the class labels. This soft voting returns the the class label as the argmax of predicted probabilities. Specifically, we assigned weights of ⅛ to K neighbours classifier, ⅛ to Random Forest Classifier, and ¾ to Linear Discriminant Analysis. The weights parameters were chosen based on hand-tuning. After assigning the weights, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The class label with the highest probability is chosen as the final label.

$$P(class_i) = P(class_i|KNB) \times w_1 + P(class_i|RFC) \times w_2 + P(class_i|LDA) \times w_3$$

Results: As we can see from the table below, the voting classifier successfully achieved a lower log loss than any of its voting components. The voting method achieves a log loss of 0.63474 on the testing data set.

	Accuracy	**Log Loss**
K Neighbours Classifier	88.89%	1.5755
Random Forest Classifier	88.89%	1.2929
Linear Discriminant Analysis	97.98%	0.9302
Weighted Average Probabilities	95.45%	0.6060

Further Improvements: The parameters of the weighted average probabilities and its three voting components can be further tuned using loops. The resulted log loss can be plotted to

identify the optimized parameters. However, this may over complicate our model, which may result in overfitting of the training data.

Linear discriminant analysis (Minghong Gabriel Zhou)

Motivation: Linear Discriminant Analysis is an classic classifier. It suggests a linear decision surface. It was chosen over other classifiers, because it works well in practice and have no hyperparameters to tune. It is relatively convenient to use compared to other classifiers with parameters that requires tuning.

Method: It is extremely easy to apply Linear Discriminant Analysis. Since there is no requirement for tuning the parameter. The algorithm can be used directly.

Code for import:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
clf = LinearDiscriminantAnalysis()
```

Instead of explaining how I prepare the data, I will explain the rationale behind this algorithm. The equation for linear discriminant analysis can be derived from predictions obtained by using Bayes' rule.

$$P(y=k|X) = \frac{P(X|y=k)P(y=k)}{P(X)} = \frac{P(X|y=k)P(y=k)}{\sum_l P(X|y=l)\cdot P(y=l)}$$

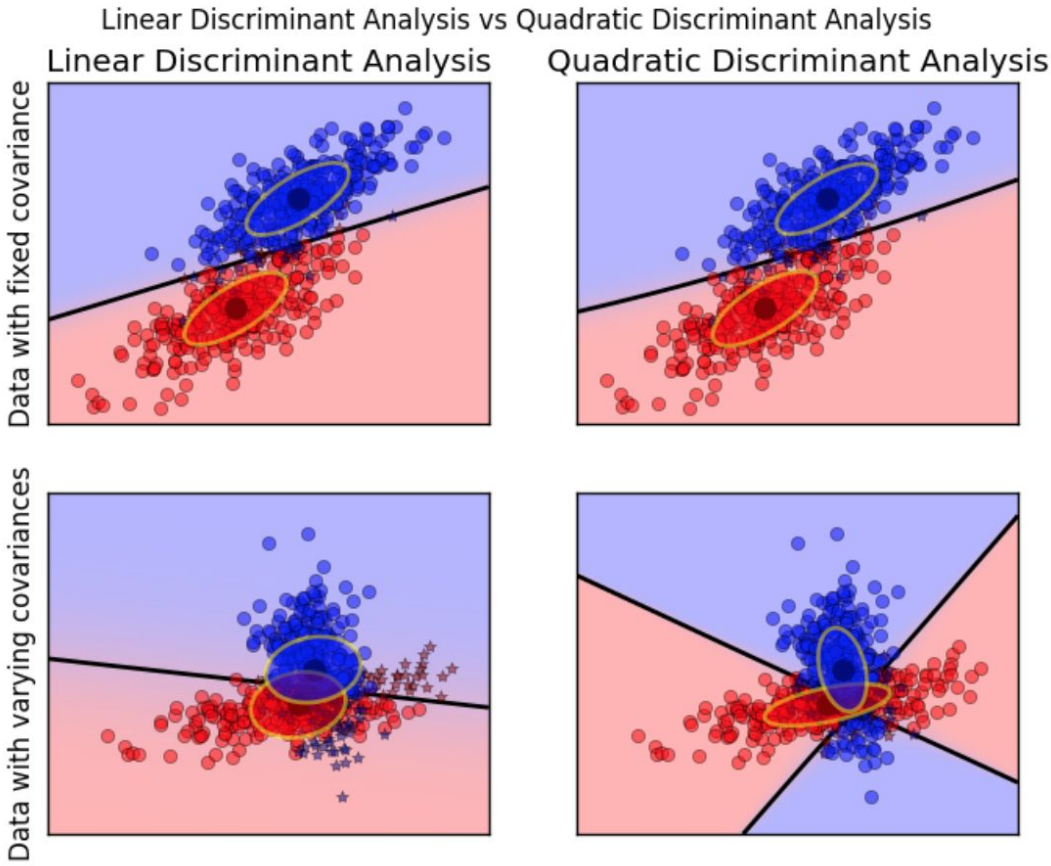
We select a k that maximizes the conditional probability.
For linear discriminant analysis, P(X|y) can be modelled as a multivariate Gausssian distribution with density:

$$p(X|y=k) = \frac{1}{(2\pi)^n|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu_k)^t\Sigma_k^{-1}(X-\mu_k)\right)$$

Result:
==========
LinearDiscriminantAnalysis
****Results****
Accuracy: 97.9798%
Log Loss: 0.930197776314

The result is desirable. With Linear Discriminant Analysis, we can achieve an accuracy of 97.9798% and a log loss of 0.930197776314.

Future work:



The above graphs show how Linear Discriminant Analysis and Quadratic Discriminant Analysis perform on the same sets of data. As we can observe from the result, the Linear Discriminant Analysis is only capable of learning linear boundaries, while Quadratic Discriminant Analysis can learn quadratic boundaries. Some inaccuracy might arise from this boundary inflexibility of Linear Discriminant Analysis.

Gradient Boosting Classifier (Linxi Han)

Boosting is a technique working on the principle of ensemble. It combines a set of weak learners to produce prediction with more accuracy. It can be used both for regression and classification. GBRT has the following advantages [1]:

- Natural handling of data of mixed type (= heterogeneous features)
- Predictive power
- Robustness to outliers in output space (via robust loss functions)

The algorithm works like this [2]:

Input: train set $\{x_i, y_i\}_1^n$, loss function $L(y, F(x))$, times of iterations M,

1. Initialize the model with a constant value: $F_0 = arg\ min \sum_1^n L(y_i, \gamma)$.

2. For m = 1, 2…,M, Compute *pseudo-residuals*: $r_{im} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$ where $F(x) = F_{m-1}(x)$ for all i; then fit a base learner to *pseudo-residuals*; compute multiple to solve the problem: $\gamma_m = arg\ min \sum_{1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$; update the model with γ_m .

3. Output F_m.

We used GBRT with Python library scikit-learn to see the accuracy of classification and tried to promote this accuracy by tuning the parameters. The parameters of GBRT includes tree based and boosting parameters. Tree-base parameters include min_samples_split, min_samples_leaf, min_weight_fraction_leaf, max_depth, while boosting parameters include learning_rate, n_estimators, etc. The parameter-tuning method can be approached in this order [3]:

1. Choose a relatively high learning rate: learning rate is better when it's low because low learning rate can avoid overfitting, but in the first step a relatively high learning rate can save the computation consumption;
2. Determine the optimum number of trees: the number should range 40-70;
3. Tune the specific tree based parameters;
4. Lower the learning rate.

The result of accuracy was range from 30%-68%. The best accuracy we can get is when n_estimators = 60, learning_rate = 0.1, max_depth = 7, random_state = 0, as following:
accuracy: 67.1717%, log loss: 2.67299265983325.

Adaptive boosting(Lingzhi Li)

Adaptive boosting is popular used for classification for combining weak learners into one strong learner. It proved to be a very successful algorithm for the two-class problem[1] with little influence by the overfitting effect. Many techniques were raised afterwards in order to work multi-class problem based on AdaBoost including SAMME and SAMME.R, which are widely accepted and implemented for extending the original algorithm

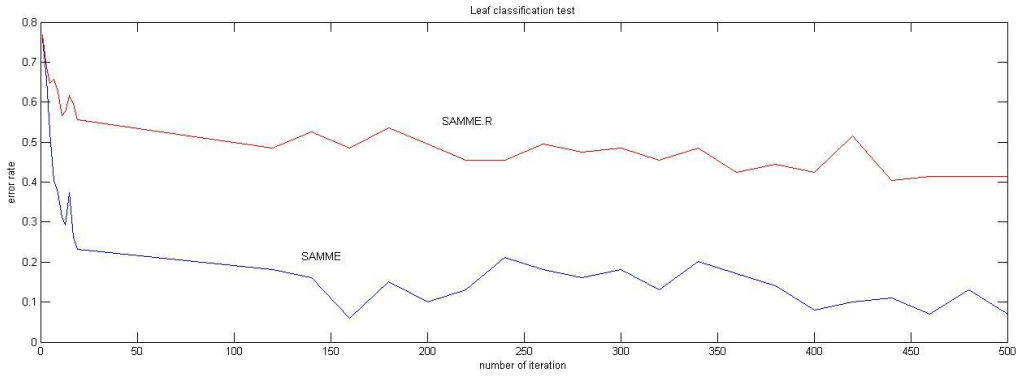
Leaf classification is often a problem with a bunch of features. AdaBoost would be a strong tool for leaf classification because, just as we mentioned, it can avoid overfitting. However, some revisions should be made to the original question so that requirements of AdaBoost algorithm can be satisfied.

The first, and also, the most important problem is how to choose adequate weak learners. The SAMME.R requires that the performance of one single weak learner be larger than $\frac{1}{K-1}$ where K is a well chosen parameter for the error term:

$$\alpha^{(m)} = \log\frac{1-err^m}{1-err^m} + \log(K-1)$$

Hence we can choose a very simple classifier for weak learners such as decision tree with small max depth at the cost of more iteration times. Or we can also choose a decision tree with higher max depth so fewer iterations are needed for a good result.

Using SAMME and SAMME.R, I have tested the performance w.r.t. different iteration times shown as below:



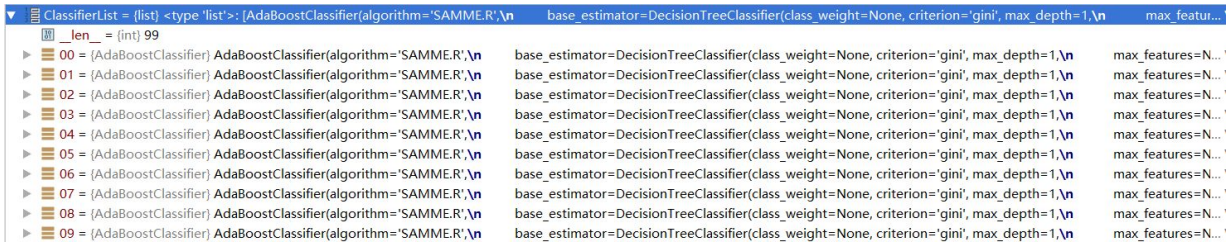
Another way for applying the AdaBoost is to reduce multi-class problem to several two-class problem.In this case, even decision tree of max depth of 1 can be used as weak learner.However we have to make more trees to make this model work.My general design idea is:

1.Start from a very few iteration times(say 3)

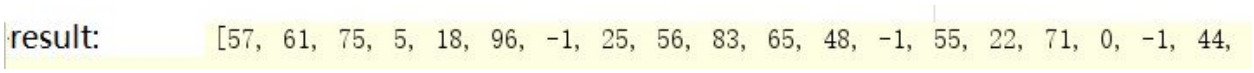
2.Combine N features in the train set into 2 classes.A simple way is to choose one of the features as one class and others as the other.Apply Adaboost algorithm with depth=1 decision tree and we can obtain the result from the test set.

3.If the result is better than the threshold I set previously ,then that classifier can be used for classify this certain feature.
4.Apply (2) and (3) for all the remaining features need to analyze and obtain all the classifiers that satisfy the threshold.
5.Increase the iteration times and repeat (2)(3)(4) until all the features have its own classifier.

The one-feature recognize classifiers are combined together to make a complex one(shown as below) used for distinguish all the feature.



Classifiers that come first are generated with fewer iteration times and can be treated to have more obvious properties to tell.Test result are shown as below with -1 denotes unable to class.



Using method above,max iteration times is only 20 and the accuracy can be 0.795.Which can be definitely improved by increasing the iteration times.

References

https://www.kaggle.com/jeffd23/leaf-classification/10-classifier-showdown-in-scikit-learn/discussion
This reference was vital to our project as it helped us get our data loaded and formatted properly. It also gave us a good baseline to decide which ensembles to pursue further

http://www.exegetic.biz/blog/2015/12/making-sense-logarithmic-loss/

http://scikit-learn.org/stable/modules/lda_qda.html

https://en.wikipedia.org/wiki/Random_forest

[1] http://scikit-learn.org/stable/modules/ensemble.html#classification
[2] Hastie, T.; Tibshirani, R.; Friedman, J. H. (2009), *The Elements of Statistical Learning*.
[3]https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/