Mushroom Classification

Tristan J. Hillis, Beverly Hom, Cody Jordan, Yogesh Kohli

Abstract

Various machine learning algorithms can be used to classify a set of examples. When picking an algorithm, one should pick in accordance with the data they are making decisions on. We chose to compare multiple learning algorithms for classifying mushrooms as poisonous or edible. Beginning with a list of ~8,000 samples of mushrooms with various attributes. The main goal of our endeavors was to determine whether or not a mushroom would kill you. Given death is an extreme consequence for guessing incorrectly, we hoped to find a learning algorithm which would give us the least error.

In this report we will discuss the Logistic Regression and Random Forest ensemble learning technique. We will then look at implementations of each and show a bit of the final product of each algorithm. This will be followed by graphical analysis of the algorithms. And finally, the algorithms will be rated and judged accordingly.

Introduction

Delicious, deadly, magical, intoxicating, mysterious. Throughout history mushrooms have gained many varying reputations, considered both food and foe. Today it is easy for us to find safe, tasty mushrooms at the grocery store, but it wasn't always this way. Over the years reckless mushroom hunters have thrown caution to the wind with sometimes fatal results, giving food-safe mushrooms a bad reputation.

Mushrooms are often lumped into the vegetable category, though most of us know that they are actually a fungus. Today the most commonly consumed variety is the button mushroom, or *Agaricus bisporus*, which makes up about 40 percent of the mushrooms grown around the world. The name "mushroom" has been given to over 38,000 varieties of fungus that possess the same thread like roots and cap. These threads, sometimes referred to as "gills," are responsible for giving mushrooms like portobellos their meaty taste and texture. As air passes through the threads moisture evaporates, giving the mushroom a rich heartiness you can really sink your teeth into.

For centuries relatively little was known about mushrooms, and for a long time the Eastern half of the world was considered mostly mycophilic, and the West mycophobic. This all changed when the French introduced mushrooms into their haute cuisine. It wasn't long before the rest of the world began to embrace the mushroom. By the late 19th century, Americans were cooking up mushrooms in their own kitchens. Prior to this time, mushrooms were mainly reserved for use in condiments. Inspired by the French, Americans took mushrooms to a whole new level of devotion. Clubs dedicated to foraging, identifying and cooking various varieties of fungi began popping up all over the country. Even today, locally

foraged mushrooms are worth their weight in gold… just ask any mushroom hunter in search of morels after a spring rain shower.

Data

Our data comes from Kaggle an online repository where data is posted for interested individuals to apply machine learning (ML) algorithms on (https://www.kaggle.com/uciml/mushroom-classification/data). Before Kaggle, this data was posted on the University of California Irvine ML Repository. It was donated by one Jeff Schlimmer. See references below.

Methods/Analysis

There are several ways to approach classification problems in ML with the multitude of classifiers from simple ones like Logistic Regression (LR) to the more complex such as Neural Networks. All these methods implement a different technique to achieve, effectively, the same things.

Before applying each classifier, we first scale the data making our feature set homogeneous. This is performed regardless of whether the classifier requires it. The data is then split 70-30 into training and testing data. Cross validation is then performed utilizing a KFolds algorithm on the training subset to tune our hyper parameters while mitigating overfitting. In this section, we look further into three different classifiers for the case of binary classification and attempt to quantify the best model of the three using the `scikit-learn` software package in Python.

We also wanted to visualize how correlated features were, by plotting a heatmap correlation matrix. There are 21 features plotted in the figure, because we removed “veil-class” which only had 0 Positively correlated features are indicated by a numeric value in the matrix greater than 1. Whereas the negatively correlated features are indicated by a numeric value less than 1. By plotting a correlation matrix, we can visually see the pairwise relationship between features. The feature importance is indicated by higher correlation with the target feature.

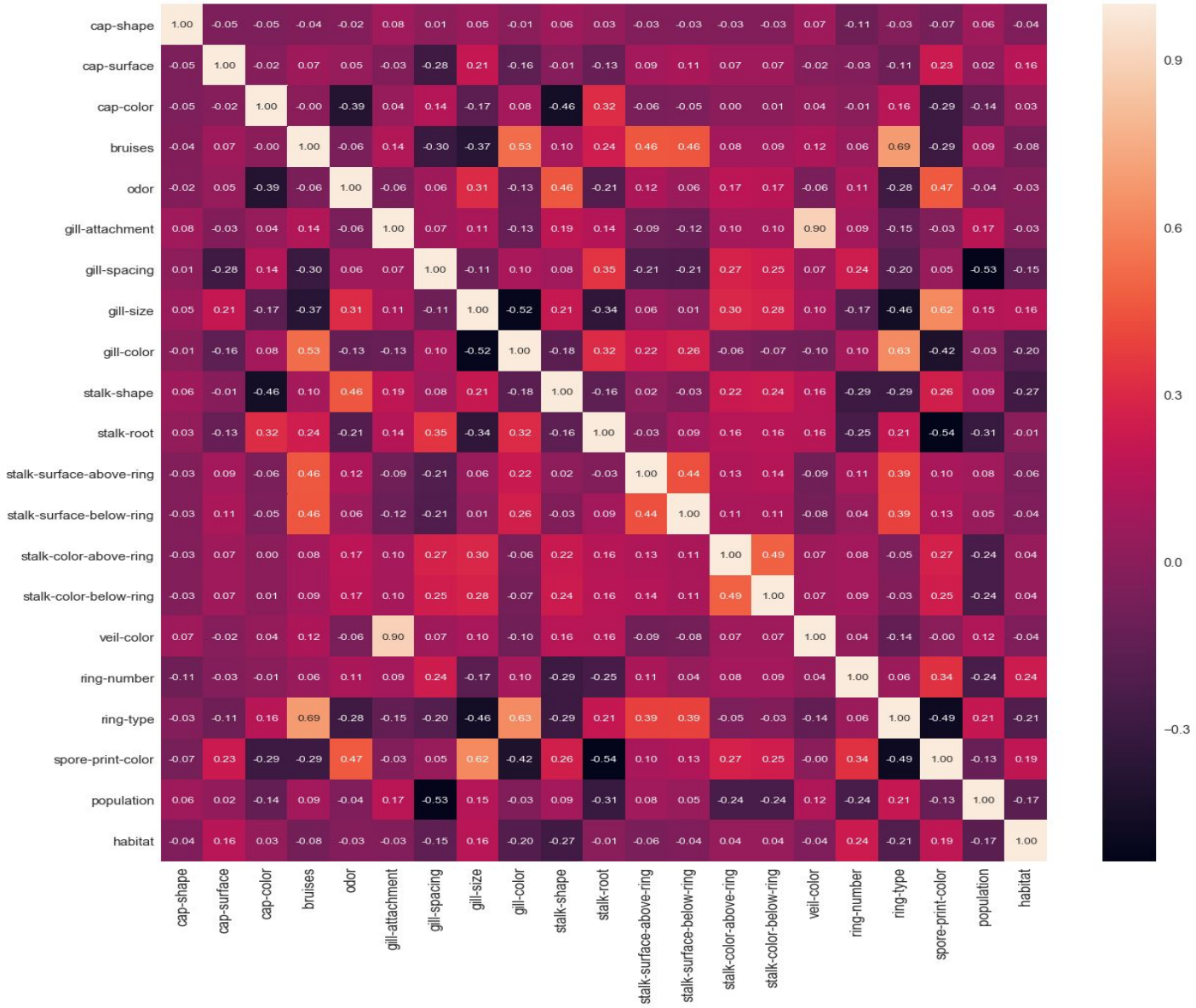
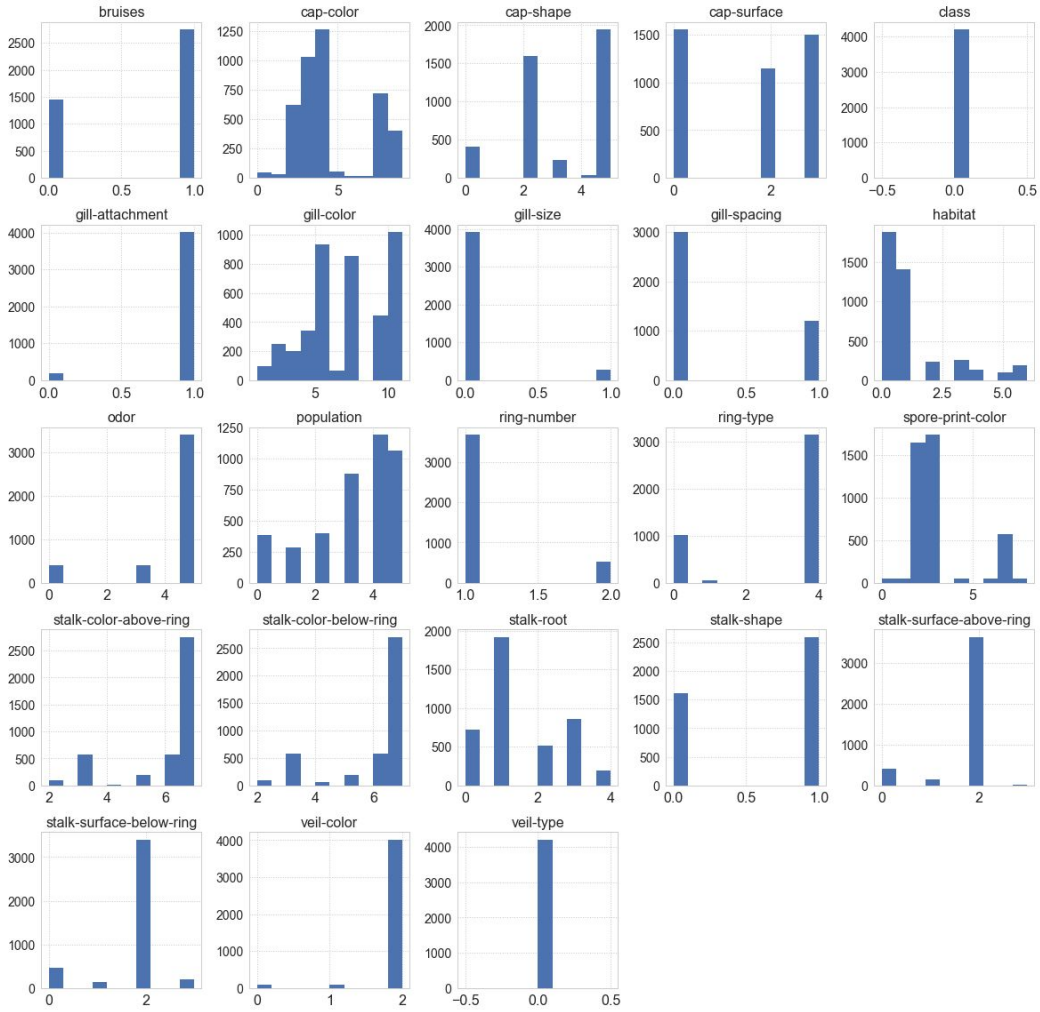
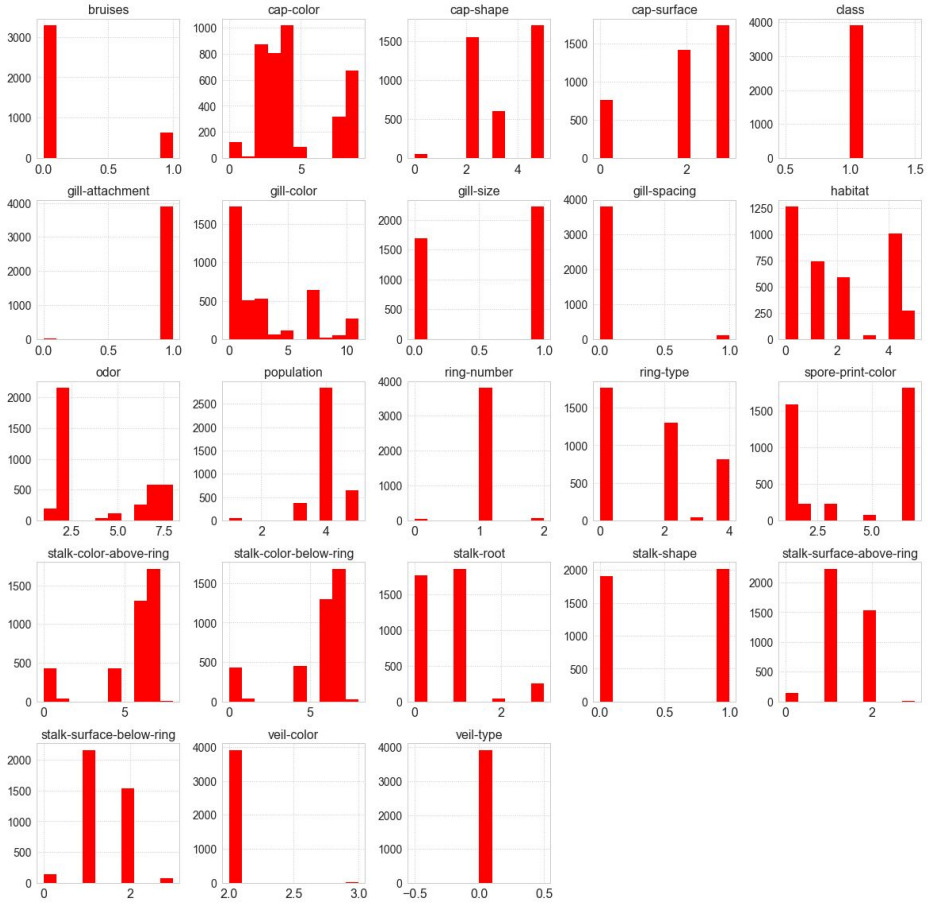


Figure _: Heatmap Correlation Matrix of Features

Distribution of Features for “Edible” Mushroom Class



Distribution of Features for “Poisonous” Mushroom Class



Logistic Regression

Usually seen as the quintessential classifier in classrooms everywhere, LR is a powerful general purpose classifier. Simple enough to code from scratch, LR sees use as a good baseline classifier. However, despite all its strengths one will typical fail to achieve the highest degree of accuracy.

For multiple features we start with putting together a linear equation of all our features in the following way: $t = \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n \rightarrow \theta^T x$. This is then put through the Logistic function which looks like the following:

$$h_\theta(x) = \frac{1}{1 + e^{-t}} = \frac{1}{1 + e^{-\theta^T x}}$$

Ultimately we can classify something as being 1 or 0 based on if the value is highly negative or positive, respectively.

We utilize the convenient function `linear_model.LogisticRegression,` which allows us to simply train the model on passed in data. After the model is trained we can pass it

the testing data to make predictions. During cross validation we tune the hyper parameter *C* which affects the strength of the *L2* regularization term in our cost function. More specifically, *C* is the inverse of this strength term that is usually denoted as λ. Smaller values of *C* mean stronger regularization giving a greater penalty to the fit, and vice-versa for larger values of C.

In the Figure below we plot the average scores from our cross-validation over various values of *C*. The scores shown are the accuracy, recall, precision, and ROC. We use all four scores to make a robust decision on the the selected *C*. The Figure shows that small *C*s do not produce good results, whereas increasing values of *C* level off. We therefore adopt the default value of *C = 1.0*, because the scores have mostly leveled off by then.

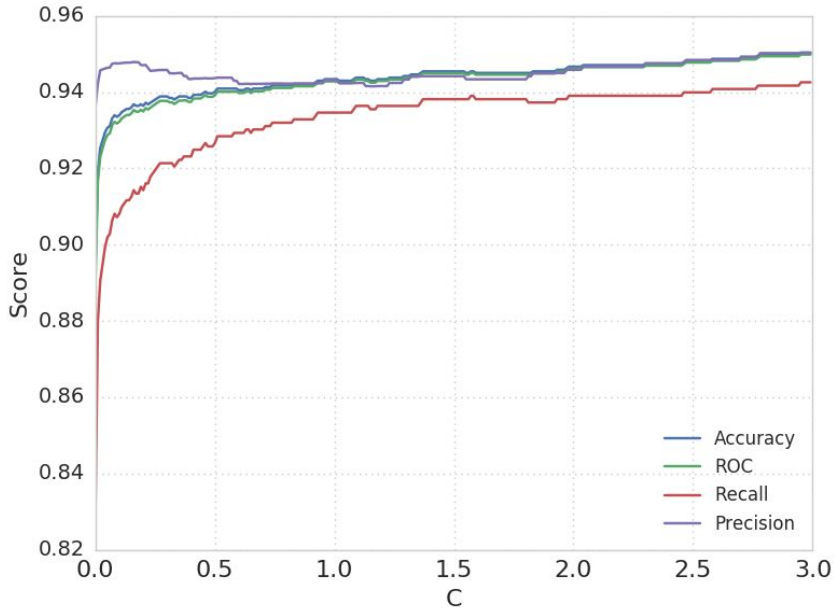


Figure: This figure shows the Stratified KFold (with 10 splits) testing with the hyper parameter *C*. We plot scores vs a changing *C* hyper parameter to determine the best value of to use for the final learning. Things level off by about ~1.0 therefore we use the default hyper parameter value of $C = 1.0$.

In addition to this cross-validation, we run another fit using only five of the highest correlated features. As seen in the figure below, because of the regression we actually lose accuracy when removing them. Like in linear regression, when we add more correlated features we become more accurate, whether for the best or not. Therefore it is best to keep all features.

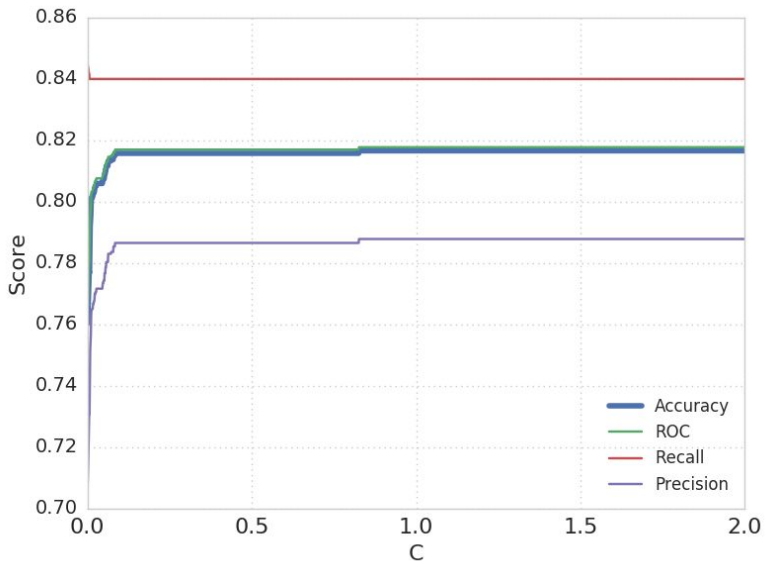


Figure: This shows that scores are worse when we only cross-validate for the five most correlated features in our set. Therefore we choose to learn using our entire data set.

K-Nearest Neighbor

K-nearest neighbor is a nonparametric method that can be used in both classification and regression problems. For classification, the output of an element is a class membership based on its nearest neighbors. The k-nearest neighbor is referred to as “instance-based learning” with fewer hyperparameters. This algorithm is also easier to implement than other algorithms and has faster training times compared to more complex algorithms like artificial neural networks. Parameters that can be tuned are: k (number of neighbors) and distance metric (i.e. euclidean, manhattan). Generally, k should be an odd number for binary classification problems, as an even number would require some rule for a “tiebreaker”. If k is set to an even number (i.e. k = 2), an element may be evenly classified into class A and class B. Therefore leaving the element unclassified unless a rule is created.

Random Forest - Python

A random forest (RF) is named as such because it is a joint construction of multiple decision trees.
Each decision tree is constructed as a model with only a few select attributes in the data set. Our algorithm was constructed based off Ho’s formulation, which implements stochastic discrimination, allowing overtraining of the data. Instead of choosing attributes with higher gain/entropy, Ho’s method consists of randomly grabbing attributes from the set of all attributes for each tree constructed.

However, in our tests we found that overtraining caused the accuracy of the algorithm to decrease. We induced that this may be because of our lack of pruning in our trees which could

have a significant effect on the validation tests.

Although the RF lacks in accuracy and variance, it has many advantages in real world data. The RF is able to handle outliers in dataset, unknown values, mixed attributes, and large datasets. In addition, a random forest is more likely to be suited for categorical data, which is what we were given in regards to each mushroom.

There are 3 features which can be tuned to improve the performance of Random Forest

1) max_features 2) n_estimators 3) min_sample_leaf

A)max_features: These are the maximum number of features Random Forest is allowed to try in individual tree. 1) Auto : This will simply take all the features which make sense in every tree. Here we simply do not put any restrictions on the individual tree. 2) sqrt : This option will take square root of the total number of features in individual run. For instance, if the total number of variables are 100, we can only take 10 of them in individual tree. 3) log2:It is another option which takes log to the base 2 of the features input.

Increasing max_features generally improves the performance of the model as at each node now we have a higher number of options to be considered.But, for sure, you decrease the speed of algorithm by increasing the max_features. Hence, you need to strike the right balance and choose the optimal max_features.

B) **n_estimators** : This is the number of trees you want to build before taking the maximum voting or averages of predictions. Higher number of trees give you better performance but makes your code slower. You should choose as high value as your processor can handle because this makes your predictions stronger and more stable.

C)**min_sample_leaf** : Leaf is the end node of a decision tree. A smaller leaf makes the model more prone to capturing noise in train data. Hence it is important to try different values to get good estimate**.**

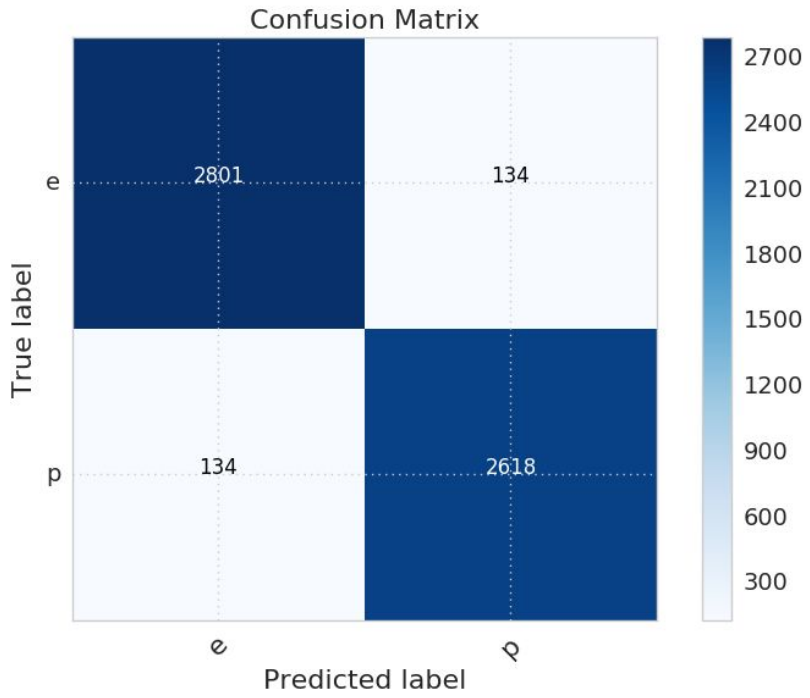
Random Forest - Scala

For the Spark-Scala implementation of RF, we choose a somewhat simple path for determining variable importance. Put simply, we created a model given a single variable and the classification. We thought this would give us a rough estimate of the importance of each variable because we could test how well the model did without the influence of the others. The data was split 70% training and 30% testing sets. We used a Spark ML CrossValidator with a RandomForestClassifier with 10 folds cross validation method. Once we obtained all accuracies for each variable we constructed a plot detailing the order of accuracies. Finally, we ran the same classification again with all variables that showed a 80% accuracy or above to obtain the final results for the Spark-Scala implementation which was odor and spore.

Results

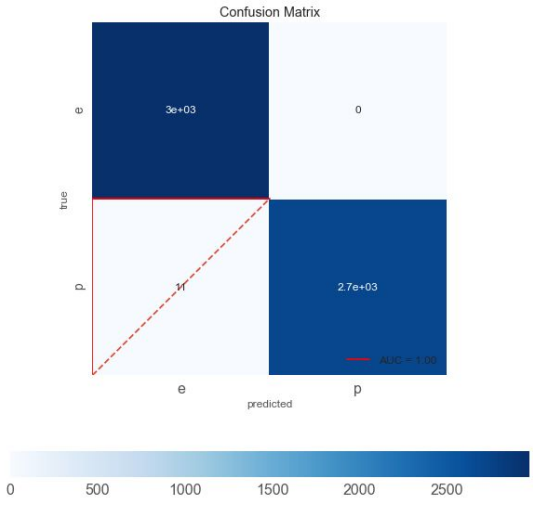
Logistic Regression (Tristan)

Although the LR is easy to implement and therefore broadly usable, we manage to get a relative good accuracy of ~95% using all features. Below is a plot of our LR confusion matrix.

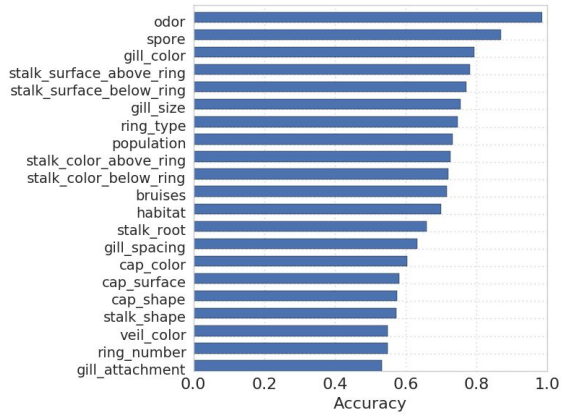


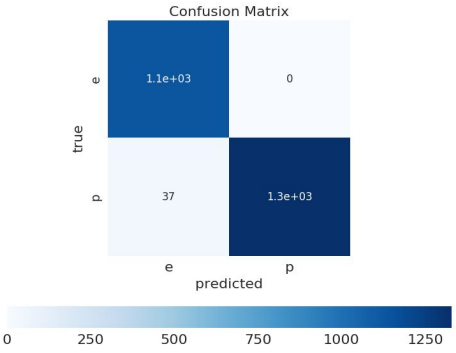
Random Forest in Python (Yogesh)

```
Accuracy is :
 0.998065764023
Confusion Matrix  is :
[[2968    0]
 [  11 2708]]


AUC_ROC IS :
             precision    recall  f1-score   support

          0       1.00      1.00      1.00      2968
          1       1.00      1.00      1.00      2719

avg / total       1.00      1.00      1.00      5687


F1 Score  is : [ 0.99815033  0.9979731 ]
Best Score :  0.9971276159212146


Best Parameters - Hyperparameters {'n_estimators': 160, 'min_samples_leaf': 10, 'max_features': 'auto'}


Train score:
 [ 1.          1.          1.          0.98275862  0.98333333  1.          1.
  1.          1.          1.        ]

Training Accuracy: 1.00 (+/- 0.01)


Test score:
 [ 1.          0.96825397  1.          0.96666667  1.          1.
  0.98387097  0.96774194  0.98333333  0.98333333]

Test Accuracy: 0.99 (+/- 0.03)
```



The first figure shows the result of the RF Classification when applied to all the features. The Training Accuracy and Testing Accuracy comes out to be 100% and 99% (+ - 0.3%) respectively. The second figure shows the confusion matrix for a RF Classification in Python with all the features.

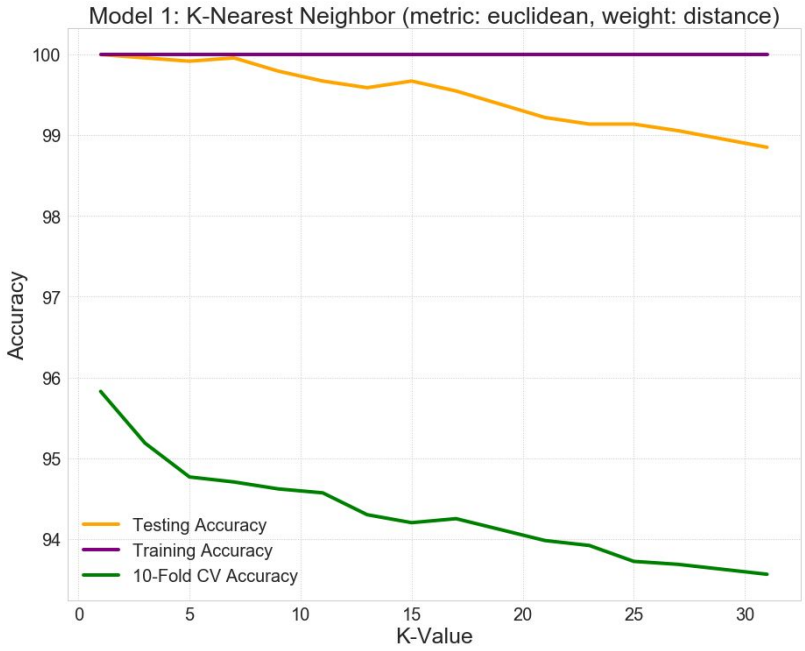




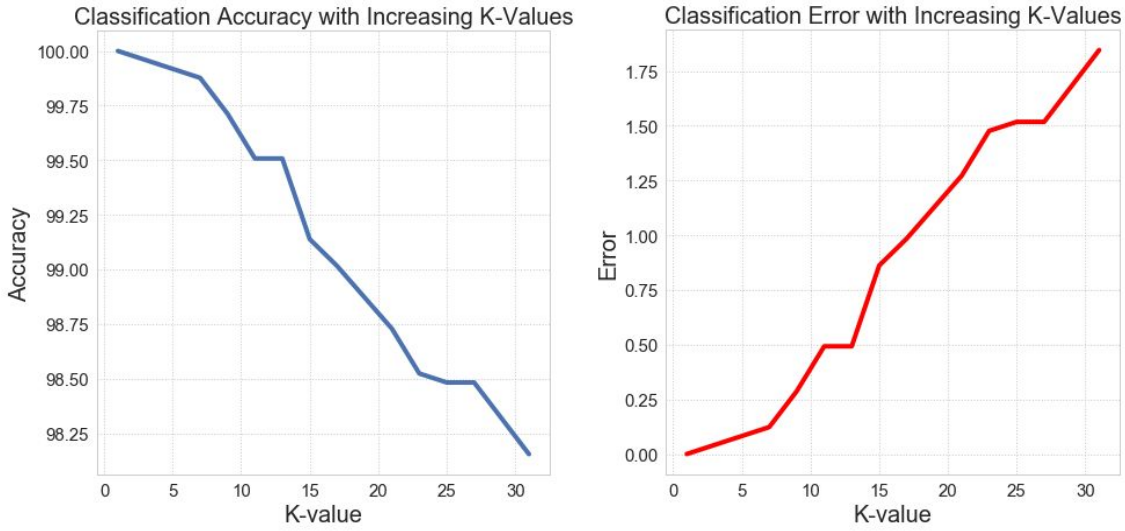
The figure on the above left show all variables ranked in terms of their individual accuracies on a RF classification. The figure on the above right shows the confusion matrix for a RF classification with two input variables, odor and spore. The accuracy, error rate, false positive and false negative percentages come to: 98.5%, 1.5%, 0% and 2.7% respectively.

K-Nearest Neighbor (Beverly)

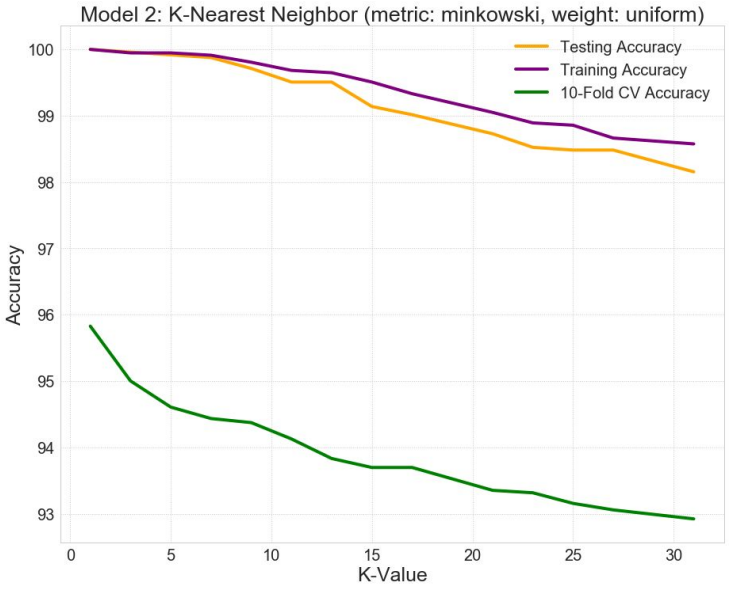
Three types of k-nearest neighbor models were tested against each other to assess training, testing, and cross-validation accuracy. Every model was tested using the same set of k-values. Specifically, odd k-values from 1-31 were tested across 3 models. Odd numbers were only chosen because of the binary classification.



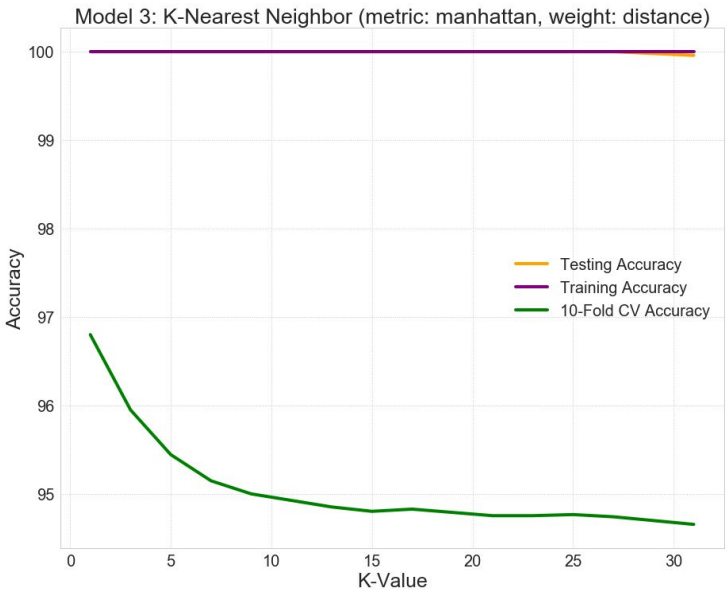
This figure shows the results from Model 1 with euclidean distance metric and a distance-based weight. As the k-value increases, the accuracy decreases across training, testing, and the validation data points. The best k-value from this model would be a lower k-value such as: k = 3.



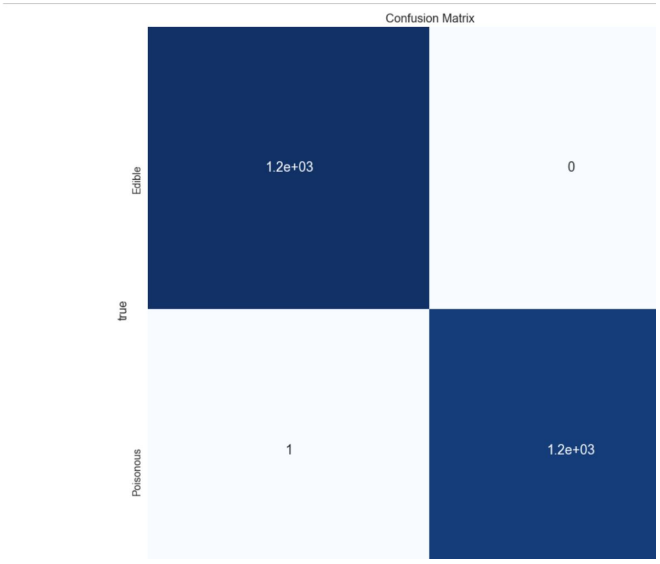
The figures above represent the classification accuracy and error of Model 1. These results illustrate that a smaller k value is suitable, and as k increases the misclassification error increases as well.



This figure shows the results from Model 2 with a minkowski distance metric and uniform weight. Model 2 has a similar trend to Model 1 in terms of testing and validation accuracy. A small k-value such as k = 3 would also be appropriate.



This figure shows the results from Model 3 with a manhattan distance metric and distance-based weight. Unlike Model 1 and Model 2, the training and testing accuracy in Model 3 overlap. This indicates that the model can be overfitting the data and memorizing the training data rather than learning. The cross-validation accuracy is similar to the previous models.



The confusion matrix represents the best prediction results from Model 1. None of the edible mushrooms were misclassified, but one of the poisonous mushrooms were classified as illustrated by the confusion matrix.

Discussion / Conclusion

Overall the best classifier was KNN based purely off the confusion matrix. However, both RF classifier implementations were close behind. The logistic regression fails to make it to the 99th percent classification accuracy. Furthermore, the speed of KNN is reasonable, whereas the RF classifier takes awhile, but is then much faster in the Spark/Scala implementation.

The use of Spark and Scala (Cody)

We decided to investigate a Spark-Scala implementation of RF classification because of the advantages with using it. Most people know that Python is such a popular languages due to its flexibility and large number of libraries, including Spark. According to Apache Spark's website (citation: https://spark.apache.org/), it is "a fast and general engine for large-scale data processing". The way Spark does this is by distributing data across a cluster of computers so that data processing can be done in parallel. This allows large amounts of data to be consumed and processed faster than most single threaded applications.

Apache Spark supports multiple languages including Java, Scala, Python and R, we decided upon the Scala implementation for an exploratory look into parallelization and big data. The reason for this other than speed is that python data types must be converted into Spark types before it can be processed whereas, Scala data types are native to Spark and work more seamlessly. This reduces the time and space complexity of a Python implementation. The final results for the python and Spark-Scala implementations of RF using cross validation showed that Python took roughly 2.5 minutes, whereas the Spark-Scala implementation took ~30 seconds.

Difficulties with past data set

The original intention of this project was to tackle the difficult problem of exoplanet classification. In particular, we would utilize flux data of measured stars from the Kepler II mission. A machine learning algorithm would be put to the task to detect exoplanets and classify the star as either having or not having exoplanets. At its roots, it seems like a simple problem; detect any dips in brightness of a star and you should have a transiting exoplanet. However, a naive approach to this problem would yield less than stellar results.

The first difficulty in this classification problem is that we are utilizing time series data which requires a different approach from your typical classification problems. The second is that exoplanet transits are quick and minute events. Among the noise of measurements you might have small, fast, dips in star brightness that would go unnoticed to the untrained eye. Thirdly, stars, although modeled often as black bodies, are in fact not black bodies. There can be both intrinsic and extrinsic factors that vary

stellar brightness wildly. For example, some stars have a pulsating outer atmosphere that vary the measured flux. Other stars are a part of a system of two or more bodies that eclipse each other affecting the flux. All these issues combine to make a truly difficult classification problem that is still a current issue among Astronomy research.

References

Mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf

Schlimmer,J.S. (1987). Concept Acquisition Through Representational Adjustment (Technical Report 87-19). Doctoral disseration, Department of Information and Computer Science, University of California, Irvine.

Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.