Udacity

Machine Learning Nanodegree

Mountain View, CA 94040

Capstone Project Report

Mushroom Identification in the Agaricus and Lepiota Genera

Raymond W. Holsapple

raymond.holsapple@gmail.com

Submitted January 29, 2018

1 Definition

1.1 Project Overview

My proposed project lies within the domain of the biological sciences. I want to design a supervised learning algorithm that identifies whether or not a mushroom sample[1] is edible or inedible. Humans have made use of mushrooms as a significant source of food, medicine, and altered awareness for many thousands of years. Hundreds of books have been written on mushroom identification and cultivation, e.g., [1, 2, 3, 4]. With modern advancements in bioinformatics and medical technology, the opportunity to discover and harness the nutritional and medicinal benefits of fungi has never been more relevant. As such, wild mushroom collecting is growing in popularity. Social media, dedicated websites, and printed field guides contribute to the rise in accidental poisonings, which may or may not be fatal. According to Tom Oder [5], misidentifications are a symptom of "knowing just enough to be dangerous." I believe that the power of machine learning presents an opportunity to improve mushroom identification for the average forager.

I have a personal interest in this topic. My home is in a rural area of the southern Appalachian Mountains on 40 acres of woodland forest. Hundreds of fungi species populate my property, and I personally own the two books long hailed as the definitive texts of mushroom cultivation [1, 2].

The dataset I am working with may be downloaded from the UCI Machine Learning Repository [6] or Kaggle [7]. It was donated to the repository by Jeff Schlimmer in 1987 and includes descriptions of 8124 hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota genera. Information used to construct the dataset originates from the National Audubon Society Field Guide to North American Mushrooms ("The Guide") [3]. The Guide clearly states that there is **no simple rule** for determining the edibility of a mushroom specimen based on its observable physical characteristics; other sources [1, 2, 5] agree with this statement.

This dataset provides exactly the kind of information needed to solve the problem stated below. It has been analyzed in no less than 50 academic publications spanning a variety of supervised learning approaches, e.g., neural networks [8], nearest neighbors [9], support vector machines [11], naive Bayes [12], and instance-based classifiers [10]. Hence, I believe this dataset is appropriate to solve the problem I am investigating.

1.2 Problem Statement

The proposed problem can be stated succinctly. Given a collection of labeled (*edible* or *inedible*) mushroom data consisting of observable physical characteristics, create and train a classification model that achieves a high level of accuracy and recall when evaluated on a previously unused set of test data.

After preprocessing the dataset, I will pass the training data through the model evaluation pipeline that I will code. I will compare the performance of several classifiers initially, e.g., a support vector machine, a neural network, and several ensemble classifiers. Next, I will conduct a feature selection process so that I can compare a reduced feature model to the reduced feature benchmark model. The *feature_selection* module in *sklearn* provides a number of very good options, e.g., *VarianceThreshold*, *SelectFromModel*, and *SelectKBest* which incorporates a user selected scoring function. I will examine a range of reduced feature sizes to help determine which classifiers perform the best using limited features. I will also produce visualizations to view the results of this feature reduction process. Figure 1 summarizes the project design.

The next step will be model tuning, which will require the use of *GridSearchCV()* in *sklearn*. However, if I choose the neural network for further investigation, I will experiment with various network architectures, loss functions, and optimizers in *Keras*. The benchmark classifier described in Section 2.4 developed its logical rules using a backpropagation network. After parameter tuning is completed, I will compare the chosen model to the benchmark using the metrics described below.

[1] Title Page Image: *Lepiota castanea* - chestnut dapperling (http://www.granadanatural.com/ficha_hongos.php?cod=393)

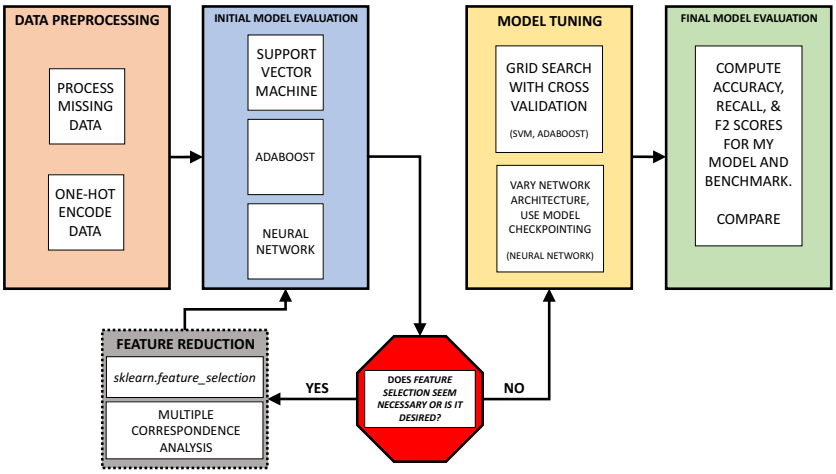


Figure 1: Capstone project workflow

A solution to this problem will be a robust classifier that accepts mushroom attributes as input and returns a label of either *edible* or *poisonous*. By robust, I mean that the classifier should perform comparably well to the benchmark model and also perform well in a sensitivity analysis. Feature reduction is an important aspect of any potential solution. Any model that needs to use all 22 mushroom attributes in order to obtain high levels of reliable prediction will be unacceptable. An acceptable solution must use only a few features, similar to the benchmark model, and still achieve high levels of accuracy, precision, recall, and F_2. Specifically, I expect recall to be very close to (or above) 0.98.

1.3 Metrics

As mentioned above, this is a binary classification problem. A true positive corresponds to labeling a poisonous/inedible mushroom as *poisonous*, and a true negative corresponds to labeling an edible mushroom as *edible*. Let the total number of true positives be P_t, the total number of true negatives be N_t, the total number of false positives be P_f, and the total number of false negatives be N_f. Using these variables I can define the three metrics that will be used to evaluate my model and compare it to the benchmark.

The primary evaluation metric is recall (R). It gives the proportion of all poisonous/inedible mushrooms that were labeled as *poisonous*. The formula is:

$$R = \frac{P_t}{P_t + N_f}. \tag{1}$$

I also want a low number of false positives; otherwise, I could label every sample as *poisonous* and have perfect recall. Hence, I want the model's precision (Z) to be high as well. The formula is:

$$Z = \frac{P_t}{P_t + P_f}. \tag{2}$$

In Equation (2), it's clear that as P_f approaches zero, Z approaches one. Thus, a model that achieves high precision is one that admits few false positives. We can combine Equations (1) and (2) in such a way that we are able to give more weight to either recall or precision. We do this with the F_β formula, given by:

$$F_\beta = \left(1 + \beta^2\right) \frac{ZR}{\beta^2 Z + R}. \tag{3}$$

If we let $\beta = 2$, more weight will be placed on recall, as desired. Hence, I will also use F_2 for evaluation. The metrics that will be used for model evaluation and benchmark comparison are accuracy, precision, recall, and F_2. Since this project is a binary classification problem, these are the most appropriate metrics.

2 Analysis

2.1 Data Exploration

Each data point in the dataset consists of 22 observable, physical attributes of a mushroom sample and its environment. The labels for the dataset are *edible* and *poisonous*. All data is categorical, and the number of categories ranges from two to twelve across the 22 attributes. One feature, STALK-ROOT, is missing 30.5% of its data; no other features have missing data. In Section 3.1, I discuss how I deal with this missing data and also the categorical nature of the data. As a representative sample, consider the following data point:

- **LABEL = POISONOUS:** CAP-SHAPE = FLAT, CAP-SURFACE = SCALY, CAP-COLOR = WHITE, BRUISES = TRUE, ODOR = PUNGENT, GILL-ATTACHMENT = FREE, GILL-SPACING = CLOSE, GILL-SIZE = NARROW, GILL-COLOR = WHITE, STALK-SHAPE = ENLARGING, STALK-ROOT = EQUAL, STALK-SURFACE-ABOVE-RING = SMOOTH, STALK-SURFACE-BELOW-RING = SMOOTH, STALK-COLOR-ABOVE-RING = WHITE, STALK-COLOR-BELOW-RING = WHITE, VEIL-TYPE = PARTIAL, VEIL-COLOR = WHITE, RING-NUMBER = ONE, RING-TYPE = PENDANT, SPORE-PRINT-COLOR = BROWN, POPULATION = SEVERAL, HABITAT = URBAN.

The data values themselves are not actually descriptive words; rather, they are single letter unique identifiers. Usually, this unique identifier is the first letter of the corresponding description, but not always. For example, in the feature GILL-COLOR, W indicates WHITE; however, in the feature HABITAT, W indicates WASTE. Moreover, HABITAT has a second category that begins with the letter "w", WOODS, which uses the unique identifier D. The complete legend of unique identifiers may be viewed on the UCI webpage [6].

According to The Guide [3] (and correspondingly [6]), There are 126 total categories across the 22 features; however, only 117 of these are expressed in the dataset. If we consider only these 117 categories, then there are approximately 1.2×10^{14} possible feature combinations. Hence, the dataset is a very small subset of all possible combinations, $6.66 \times 10^{-9}\%$ to be exact. Moreover, the distribution of the data within each feature is far from uniform, as will be explored in Section 2.2. Despite the relatively small sample size, the information embedded within the dataset is more than sufficient (and sometimes redundant) to recover the labels accurately. This will be addressed further in my free form visualization discussion is Section 5.1.

2.2 Exploratory Visualization

Even though all of the data is categorical, it is always useful to be cognizant of how your data is distributed across its domain. Presumably, the dataset is representative of the distribution of these feature categories within the genera Agaricus and Lepiota. Otherwise, the final model would likely encounter difficulty achieving acceptable levels of accuracy and recall when used to predict labels for a separate test dataset that represented realistic categorical distributions. In order to visualize these distributions, I have plotted them for all 22 features and the labels in Figure 2. Obviously, the categories are unordered, so I have not artificially arranged them along any of the x-axes. They appear as determined by Pandas when I collected the data statistics.

Upon inspecting Figure 2, the most obvious observation is that almost none of the features are close to being uniformly distributed across its categories. The two exceptions are LABEL and VEIL-TYPE. For LABEL, almost 52% of the data points are labeled *edible*, with just over 48% *poisonous*. For VEIL-TYPE, only one of the two possible categories listed on UCI [6] even appears in the dataset; all 8124 data points have VEIL-TYPE = PARTIAL. Several of the other features (GILL-SPACING, VEIL-COLOR, RING-NUMBER, GILL-ATTACHMENT) have at least 75% of their data concentrated in a single category.

There are several especially interesting results. For example, essentially all samples have a VEIL-COLOR of WHITE. Also, about 90% of the samples have a RING-NUMBER of ONE. Another interesting observation is that, despite nine categories for STALK-COLOR, approximately 75% of the samples are either WHITE or PINK. Of course, considered alone, categorical distributions reveal little insight into the relationship between a feature and the labels. For example, my project will show that the feature ODOR is very highly correlated with the labels, but it's distribution doesn't suggest this.

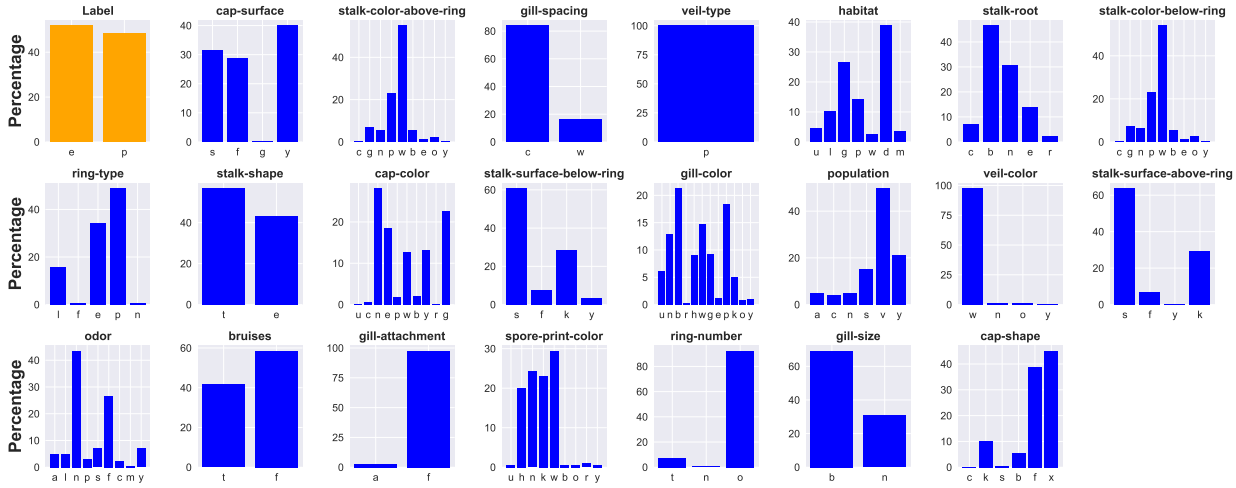


Figure 2: Feature distributions as a percentage of the total number of data points (8124)

2.3 Algorithms and Techniques

My initial model evaluation task is a standard classification on labeled categorical data. The data are relatively sparse, but not excessively sparse. After one-hot encoding, each of the 8124 data points will be a binary vector in $\mathbb{R}^{117}$ comprised of 22 ones and 95 zeros. Given the relatively small size of the dataset, I expect that any default classifier from *sklearn* ought to execute reasonably fast. As mentioned in the project overview, this dataset has been analyzed using a variety of classification algorithms [8, 9, 10, 11, 12], so I will choose eight classifiers for my initial evaluation. I used Scikit-learn's estimator-chooser webpage as a starting reference [13].

I have chosen to start with the following classification algorithms: Linear Support Vector Machine, k-Nearest Neighbors, Stochastic Gradient Descent, Gaussian Naive Bayes, AdaBoost, Random Forest, Gradient Boosting, and a Neural Network. The only default parameter I will modify initially is '*random_state*', which will be the same for all algorithms that accept such a parameter. Of course, I must be a bit more hands-on with the neural network. I will build a baseline neural network in *Keras*, which will be described in a bit more detail in Section 3.2.

Although I started my evaluation process with the eight classifiers mentioned above, three of them consistently outperformed the others throughout the refinement and sensitivity analysis stages. They are k-Nearest Neighbors, Gradient Boosting, and Random Forest. The next few paragraphs thoroughly detail how each of these three algorithms work. All information from these descriptions is found either in the MLND course or on Scikit-learn's website [13].

The k-Nearest Neighbors (kNN) algorithm is among the simplest of all machine learning algorithms. It is a non-parametric, instance-based learner where classification is only conducted locally and is deferred until the prediction stage. The training stage consists only of storing the feature vectors and labels of the training set. The 'k' in the name of the algorithm stands for the number of nearest neighbors that will be used to perform the prediction task. In other words, if $k = 3$ then kNN will find the three closest neighbors in the training set to the one for which prediction is being performed on, and then use them to predict a label for the query. The user may choose among many distance metrics within *sklearn*, but the default metric is the Minkowski metric with $p = 2$, which is simply the classic Euclidean metric. This metric is best for continuous data, but there are many great metrics that are good for categorical data. Although I started all analyses with default parameters, during parameter tuning I tested the Jaccard metric which ended up being the best. To define the Jaccard metric $J(z_1, z_2)$ for two binary vectors z_1 and z_2, let N_{tf} be the number of dimensions where z_1 is TRUE and z_2 is FALSE, N_{ft} be the number of dimensions where z_1 is FALSE and z_2 is TRUE, and

N_{tt} be the number of dimensions where both are TRUE. Then we have $J(z_1, z_2) = \frac{N_{tf} + N_{ft}}{N_{tf} + N_{ft} + N_{tt}}$. Once kNN has selected the k nearest neighbors, it must use them to make a prediction. In classification, this process is conducted via voting, with majority vote determining the classification label. If regression is being performed, kNN would compute some statistical measure of center, i.e., "average", to make a prediction. If desired, kNN also allows the user to choose how the neighbors should be weighted in the vote. The default is to use uniform weighting, but one may also choose "distance" weighting. In distance weighting, closer neighbors have more weight. kNN stands out because all of its complexity is pushed into the prediction stage. This means training is trivial, and unlike the other algorithms there is no need for sophisticated mathematical tools, e.g., calculus.

Gradient Boosting (GB) is considered an ensemble method. Ensemble methods combine predictions from a collection of base estimators to improve generalizability and robustness over a single estimator. We refer to these base estimators as weak learners. Basically, a weak learner is a learning model that is guaranteed to always do better than chance regardless of how the training data is distributed. GB uses decision trees for its weak learners. A decision tree is a simple construct. A feature and decision threshold is selected which splits the data into two groups (nodes). A label may be assigned at this point, or we could further split the data using other features and a deeper decision tree. GB supports the tuning of many parameters that control how the decision trees are constructed, e.g., maximum tree depth and minimum samples needed for a split. GB's training process begins with some initial decision tree model and continues to modify the current model additively until the algorithm reaches a pre-determined number of estimators, which may also be tuned to optimize the model. Essentially, at each stage of this additive process there are two things that must be determined: a new decision tree and a weight that will multiply the new decision tree before it is added to the previous model. The predictive power of GB lies in how these two things are computed. At each stage, the new decision tree and weight are chosen to minimize some user specified loss function. The loss function may be either "binomial deviance" or "exponential." The optimization problem is solved using the gradient descent algorithm, hence the name. Essentially, in each iteration of the optimization process GB steps in the direction of steepest descent of the loss function to minimize it as much as possible in that step. Effectively, at each stage of the process more emphasis will be given to correctly classifying points that were mis-classified in the previous step, and less emphasis will be given to points previously classified correctly. This process is repeated until the training process concludes when the maximum number of estimators is reached. Each individual estimator may only be trivially better than chance alone, but this weighted sum of estimators is much better than chance and robust against overfitting. Once the training is done, GB has a very simple and fast prediction process. This algorithm stands out from the other two because it makes use of differential calculus to fit a regression tree along the negative gradient of a loss function. It incrementally improves the model by adding new weak learners in each step. In the end, the cumulative effect is an extraordinarily robust predictor. However, the computation of so many derivatives can be costly; training can be slow.

Random Forest (RF) is another ensemble method, and it also uses decision trees as weak learners. But RF differs fundamentally from Gradient Boosting (and also AdaBoost) because RF uses an averaging method as opposed to a boosting (sequential construction) method to build a prediction model. The user chooses the number of decision trees that will be used in the forest when the classifier is initialized. Each one of these trees will be built using a random sample of the training data, with replacement unless the "*bootstrap*" parameter is set to FALSE. During the construction of each decision tree, all of the features are not used when selecting the the node that yields the best split. In fact, a random subset of the features is used when computing which node yields the best split. Similar to Gradient Boosting, the RF algorithm in *sklearn* supports the tuning of many tree construction parameters, e.g., maximum features to use when looking for the best split and minimum samples required to be in a leaf node. Two metrics may be used when computing the quality of a split: Gini impurity or information gain entropy. During parameter tuning, I tested both of these metrics and Gini impurity always resulted in the best performing predictor. After all of the random decision trees in the forest have been constructed, the bagging aspect the algorithm comes into play. During the prediction step, each of the constructed trees would make a prediction for the label of a sample. Traditionally, something similar to what happens in k-Nearest Neighbors would happen. Each tree in the forest casts a vote for the label and the majority would determine the label. However, in *sklearn*'s implementation, the label is

determined by computing the mean over all probabilistic predictions from the trees in the forest. The high level of randomness has a tendency to increase the predictive bias, but decreases the variance significantly to counteract the bias. RF stands out from the other classifiers because it incorporates the simplicity of randomness (instead of calculus) to control variance in classification. Like Gradient Boosting, RF is an ensemble method that uses decision trees, but it uses them in a far more computationally less expensive manner. Gradient Boosting's predictive power is vested in its complex design, whereas RF's power is simply randomness.

Except for naive Bayes and the neural network, all of the classification methods work superbly with sparse input data. Moreover, the ensemble methods all have several highly attractive characteristics [14]:

- work well with sparse, categorical data,
- easy to validate with statistical tests,
- robust (i.e., performs well even if assumptions are somewhat violated by true model)
- predicts quickly, $\mathcal{O}(\log n)$.

Despite a propensity for high bias associated with decision trees, I expect the ensemble classifiers to outperform the other classifiers.

A primary technique that will be implemented in my analysis is feature selection. The motivation for feature reduction is a comparison between my final model and the benchmark, which achieves very high accuracy and recall using only a few of the 22 features in the dataset. I will design a function that uses one of Scikit-learn's built-in feature selectors: *SelectKBest*. This feature selector is heavily dependent upon the scoring function used. Since, all of my data is binary (categorical) the most appropriate scoring function to use is mutual information. Mutual information measures the dependency between two variables. A score of zero implies independence; higher scores imply higher dependence.

After my initial evaluation and feature selection algorithms have completed, I will conduct a grid search to tune the parameters of my selected model(s) using *GridSearchCV()*. The model(s) that perform best in the parameter tuning will also undergo a sensitivity analysis study. This is to ensure that the results of my evaluations up through the parameter tuning have not been achieved by chance. The model should achieve high scores in all metrics even when the dataset is subjected to modest perturbation. The details of how I will perform this analysis are given in Section 4.1.

2.4 Benchmark

The dataset description on UCI [6] details benchmark results learned and shared by Duch, Adamczak, and Grabczewski [8] using a backpropagation network in 1996. These results are a set of four logical rules for identifying a sample as *poisonous*, increasing in specificity from level one (L_1) through level four (L_4). The four boolean conditionals are:

$$
\begin{aligned}
L_1 &:= \text{ODOR} \neq \text{ALMOND AND ODOR} \neq \text{ANISE AND ODOR} \neq \text{NONE},\\
L_2 &:= L_1 \text{ OR } (\text{SPORE-PRINT-COLOR} = \text{GREEN}),\\
L_3 &:= L_2 \text{ OR } (\text{ODOR} = \text{NONE AND S.S.B.R.} = \text{SCALY AND S.C.A.R.} \neq \text{BROWN}),\\
L_4 &:= L_3 \text{ OR } (\text{HABITAT} = \text{LEAVES AND CAP-COLOR} = \text{WHITE}),
\end{aligned}
$$

where S.S.B.R. is STALK-SURFACE-BELOW-RING and S.C.A.R. is STALK-COLOR-ABOVE-RING. Let X represent an arbitrary data point, and adopt the following labeling policy:

If boolean conditional applied to X is TRUE, then label X as poisonous.

Then we can compute the evaluation metrics given this policy; they are summarized in Table 1.

The benchmark policy is quite simple, using only one to six of the 22 mushroom attributes. It yields remarkably high accuracy and recall. Recall is most important due to our desire not to consume poisonous mushrooms, but a primary goal of the project is to design a classifier that correctly labels as many edible mushrooms as possible. In other words, it would not be desirable to have 99% recall at the cost of misidentifying half of the edible mushrooms.

Table 1: Metrics applied to all four benchmark levels.

Conditional	Number of Features	Accuracy	Precision	Recall	F_2
L_1	1	0.98523	1.00000	0.96800	0.97424
L_2	2	0.99385	1.00000	0.98667	0.98930
L_3	4	0.99877	1.00000	0.99733	0.99787
L_4	6	1.00000	1.00000	1.00000	1.00000

Note: the metric values from Table 1 were computed by me on the testing data alone. The accuracy numbers quoted on UCI [6] and by Duch, et al. [8] refer to this policy being applied to the entire dataset.

3 Methodology

3.1 Data Preprocessing

Some preprocessing was necessary before beginning the analysis. Approximately 1.4% of the data was missing, all within the attribute STALK-ROOT. I replaced the missing data with the unique identifier N, which corresponds to the category NONE. Next, I one-hot encoded all of the data to prepare it for input into each of the classifiers. This process increased the size of the feature space from 22 to 117. Hence, after preprocessing the dataset, I have a collection of 8124 binary vectors in $\mathbb{R}^{117}$. There is no need for any scaling or feature transformations. Feature selection/reduction evolved into a major component of this project for reasons that will become evident in the Implementation Section below. In order to make a more meaningful comparison to the reduced feature benchmark described in Section 2.4, my primary goal became the desire to design a classification model that is comparable to the benchmark in feature reduction and accuracy. Hence, I will discuss the feature selection component of my project in the Refinement Section.

3.2 Implementation

After preprocessing the data, my implementation process began by splitting the data into training and testing sets. I chose to set aside 20% of the data for testing. This set of data was not used for training or for feature selection. It was only used for prediction and evaluation. This left 6499 data points for training.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 64)                7552
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 65
=================================================================
Total params: 7,617.0
Trainable params: 7,617.0
Non-trainable params: 0.0
_________________________________________________________________
```

Figure 3: *Keras* neural network model summary

I then initialized the eight classifiers mentioned in Section 2.3. I used the default parameters, and for the classifiers that accept it I set *random_state* = 1984. In order to initialize the neural network, I had to create a baseline network in *Keras*. The baseline network consists of a '*Dense*' layer with 64 nodes and a '*relu*' activation, followed by a '*Dropout*' layer that drops 25% of the outputs. The final layer is another '*Dense*' layer with a single classifier node using a '*sigmoid*' activation. See Figure 3 for a summary of this neural network, which must fit 7617 parameters for the case of the full feature space (117 features).

I then passed the training data into the evaluation pipeline that I coded: *clf_eval_pipe()*. This pipeline relies on quite a few helper functions to process, print, and graph the results. The pipeline first calls *get_slice_limits()*

which creates a dictionary of upper boundaries corresponding to uniformly increasing data proportions. For example, if I want five subsamples the function will find the upper bound indices corresponding to 20%, 40%, 60%, 80%, and 100% of the training data. I do this so that I can create a graph of evaluation data that uses increasing amounts of the training data on all eight classifiers. Next, my pipeline determines if the user has requested to reduce the feature space. If feature selection is requested, the pipeline passes the data to the function *reduce_feature_space()*. This function uses Scikit-learn's *SelectKBest* and mutual information scoring function to choose the number of top scoring features chosen by the user.

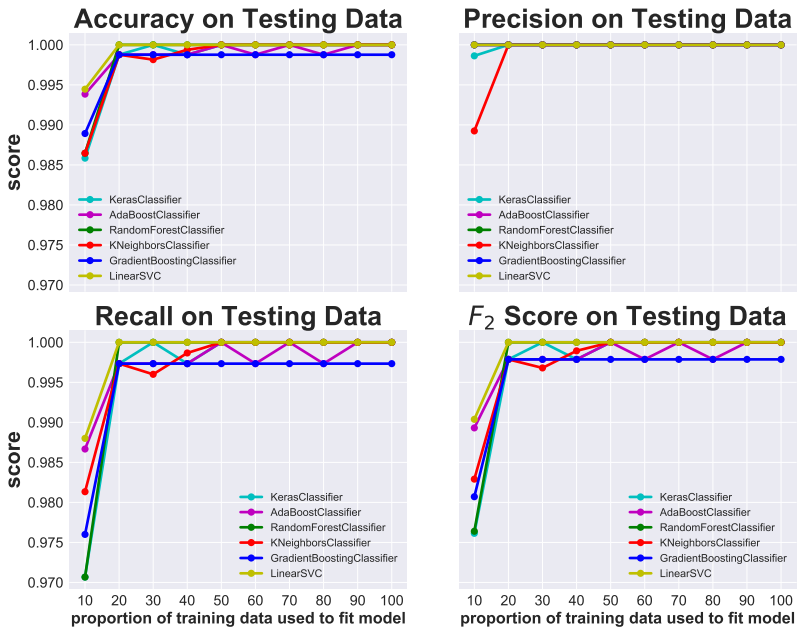


Figure 4: Initial classifier evaluation results using all 117 features

After feature reduction (if necessary), the training data and classifiers are passed to the function *build_eval_data()*. This function uses two '*for*' loops to iterate over all combinations of the classifiers and data proportions to build an entry in an evaluation data dictionary. In each iteration the function *fpe()* is called where all the heavy-lifting is performed. The function name stands for 'Fit, Predict, Evaluate.' This is where the training data is fit using the classifier, and then the testing data is used to make predictions. At that point, the predictions and the testing labels are used to compute the accuracy, precision, recall, and F_2 scores of the evaluation iteration.

Finally, after all of the evaluation data dictionaries are collected into a single *eval_data* dictionary and passed back to the pipeline function, the only thing left to do is to print and graph the results. If the pipeline's keyword argument *print_data* is True (default), the function *print_eval_data()* will be called to print the final evaluation metrics to standard out. Then, if the pipeline's keyword argument *graph_data* is True (default), the pipeline will pass the evaluation dictionary to the functions *graph_eval_data()*, *mcc_feature_heatmap()*, and *mcc_label_heatmap()*. These last two functions produce feature-to-feature and feature-to-label heatmaps, respectively, which will be described thoroughly in Section 5.1.

After passing all eight classifiers through the pipeline, I noticed that two of them (Stochastic Gradient Descent and Gaussian Naive Bayes) performed poorly compared to the other six. Hence, I re-ran the pipeline without these two classifiers. The results of this initial classifier comparison are displayed above in Figure 4.

Upon inspecting Figure 4, we notice several interesting things. Firstly, none of the classifiers seem to stand out above the others. In fact, when using all of the training data, all but one of the classifiers (Gradient Boosting) achieves perfect scores on all four metrics, and Gradient Boosting scores above 0.997 on all four metrics. Clearly, there is enough information embedded in the 117 features for most classifiers to build perfect predictor models. Another aspect that stands out is that most of the classifiers achieve this performance using only 20% to 40% of the training data. If training time were an issue, this result would support a decision to randomly subsample the training data to use during the refinement stage of the project. However, that is not the case here. The evaluation time is quite reasonable using all of the data. The only other thing to point out is the odd sawtooth behavior of the AdaBoost classifier; I don't understand it. Nevertheless, it still achieves very high performance.

I encountered very few technical complications during the coding process. Most of the complications I experienced involved the mundane but intricate details of producing publication quality images. For example, I had to develop data extraction logic to extract the data to be plotted from the data structures passed to the graphing functions. Some of these data structures were dictionaries where the keys were classifier names and the values were multi-dimensional arrays. Moreover, the data for any given *bar* or *line* in the graph was usually dispersed across all of the lists within each multi-dimensional array, which was a more convenient way of storing the data as it was created. Of course, there were also many details regarding fonts, axes labels, tick mark positions and labels, etc. that required much adjustment to make the figures look nice.

The only true technical difficulty I had was finding a clever way to vary the "*input_dim*" argument in the first layer of my *Sequential()* neural network model. I wanted my neural network to behave and be accessible in the same way as all of the *sklearn* classifiers. This would allow my *clf_eval_pipe()* function to pass training data to the *Keras* classifier inside a loop where it was also sending the same data to several other classifiers. This can be accomplished using the *scikit_learn* wrapper in *Keras*. One simply needs to create a baseline neural network model in a function, which is called by the *Keras* classifier when you initialize the classifier. I named my model (described in Figure 3) function *baseline_nn()*. The complication I encountered was figuring out the best way to declare the value of "*input_dim*" inside this function. One option was to pass it as a keyword argument to *baseline_nn()*. But in order to do this, I would have to re-initialized the classifier inside my *clf_eval_pipe()* function each time I was fitting reduced feature training data. This is because the initialization calls the *baseline_nn()* and if it's accepting a keyword argument for the input dimension size, it would require a re-initialization of the classifier with that value modified. I didn't like this option, because I desire uniformity in my code, if possible. I didn't want to initialize seven of the classifiers at the top of the notebook and then initialize the *Keras* classifier inside the evaluation pipeline. I much preferred to initialize the *Keras* classifier once in the same location as all of the others. In order to do this, I decided to introduce a global variable representing the dimension of the neural network inside *baseline_nn()* and also inside *fpe()*. Then I only needed to modify this global variable before fitting the training data in order for the first layer in my neural network model to have access to the correct size of the data input dimension size. I spent more than an hour figuring this one out. I didn't encounter any other noteworthy complications.

Moving forward I refined the results from Figure 4 by reducing the feature space in my evaluation pipeline as described above. Since the evaluation time was fast and all of these six classifiers performed well using all 117 features, I continued to evaluate all six of these classifiers on the reduced feature spaces.

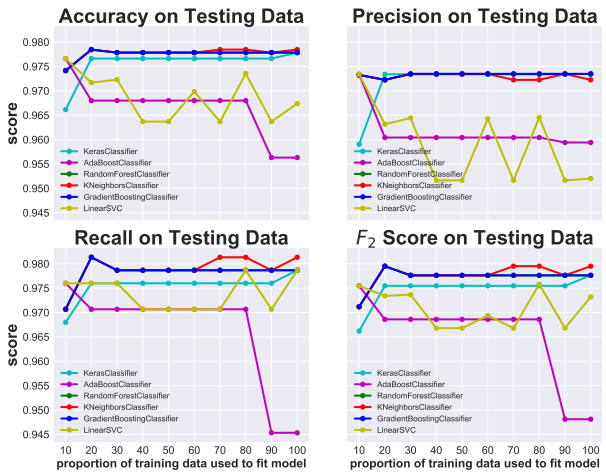
3.3 Refinement

Model refinement occurred in two stages: feature reduction followed by parameter tuning. The first priority was to reduce the feature space as much as possible while maintaining relatively high accuracy and recall scores. The primary end objective is to keep recall near or above 0.98, so the recall score will be the primary deciding factor when choosing which reduced feature space will proceed to the parameter tuning stage.

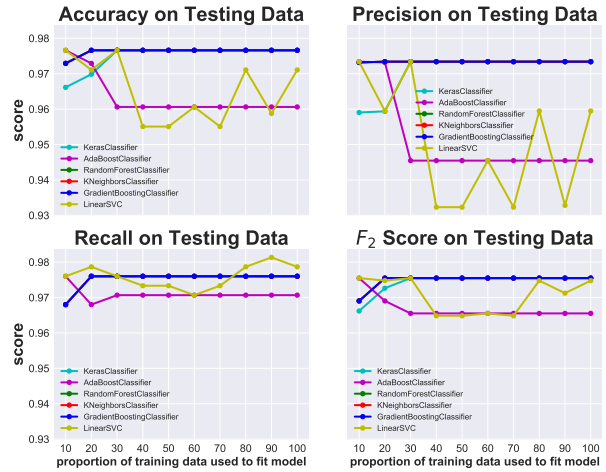
I decided to perform the feature reduction refinement using the top 15, 12, 10, 7, 5, and 2 features. Before moving forward it is important to be mindful that in my analysis, a "feature" refers to one of the results of the one-hot encoding that I did during preprocessing. This means that ODOR_F (foul odor) and ODOR_N (no odor) would constitute 2 features, even though only one of the original features is needed to uniquely determine both of the values for these constructed features. This is in contrast to the benchmark. When the benchmark refers to four features, it really means four of the original features. One of the results of my feature selection code is to return the names of the one-hot encoded features that were selected. That allows me to see how many original features (OF) are needed to determine that subset of one-hot encoded (OHE) features. These equivalences for the feature refinements mentioned above are summarized in Table 2, and the evaluation results for the first four of these feature refinements is graphed in Figure 5.

Table 2: Original features needed to determine one-hot encoded features.

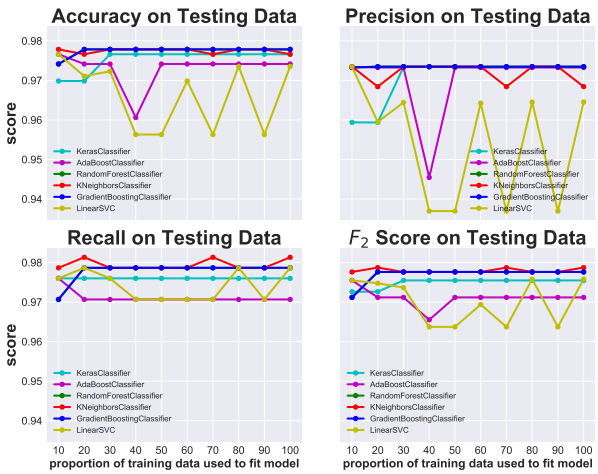
number of one-hot encoded features →	15	12	10	7	5	2
number of original features →	8	8	7	5	4	1



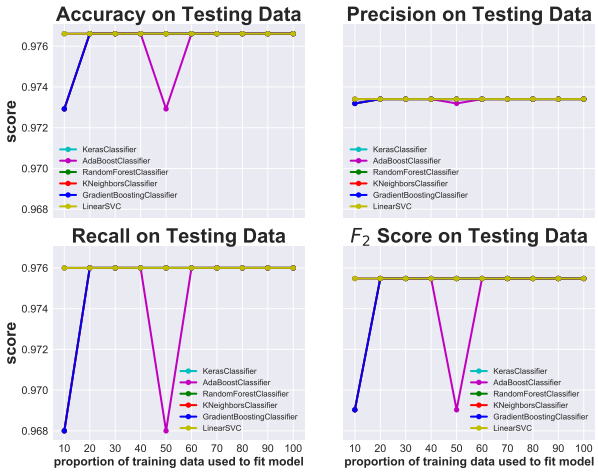
(a) Top 15 OHE features (9 OF)



(b) Top 10 OHE features (7 OF)



(c) Top 12 OHE features (8 OF)



(d) Top 7 OHE features (5 OF)

Figure 5: Classifier Refinement Evaluation Results

When inspecting the figures below, be sure to keep in mind that the y-axes are scaled differently from panel to panel. AdaBoost was consistently the poorest performer among the six classifiers during this refinement process. The *Keras* neural network, Random Forest, Gradient Boosting, and k-Nearest Neighbors classifiers were consistently the top performers. Linear SVC performed somewhat erratically in half of the graphs, but it executes very quickly and in Figure 5b, the recall score for Linear SVC was the highest. These are encouraging results. Hence, after considering these results, I decided to proceed in my model refinement process using the top seven one-hot encoded features (Figure 5d) and the four classifiers: Random Forest, Gradient Boosting, k-Nearest Neighbors, and Linear SVC. They all have identical results at this point: accuracy = 0.97662, precision = 0.97340, recall = 0.97600, F_2 = 0.97548.

```
svc_grid = {'dual': [True, False],
            'tol': [1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8],
            'C': [1, 10, 100, 1000, 10000],
            'max_iter': [1000, 5000, 10000]}

knn_grid = {'n_neighbors': [1, 2, 5, 10],
            'weights': ['uniform', 'distance'],
            'leaf_size': [1, 2, 4, 8, 15, 30],
            'metric': ['jaccard', 'minkowski']}

rfc_grid = {'n_estimators': [2, 4, 8, 10, 20, 30, 40],
            'criterion': ['gini', 'entropy'],
            'max_features': [None, 'auto'],
            'min_samples_split': [2, 4, 6],
            'min_samples_leaf': [1, 2, 4, 10],
            'max_leaf_nodes': [25, 50, 100, None]}

gbc_grid = {'learning_rate': [0.1, 0.25, 0.5],
            'n_estimators': [10, 25, 50, 100, 150],
            'max_depth': [2, 3, 5, 8],
            'min_samples_split': [2, 3, 4],
            'min_samples_leaf': [1, 2, 4]}
```

Figure 6: Parameter tuning grids

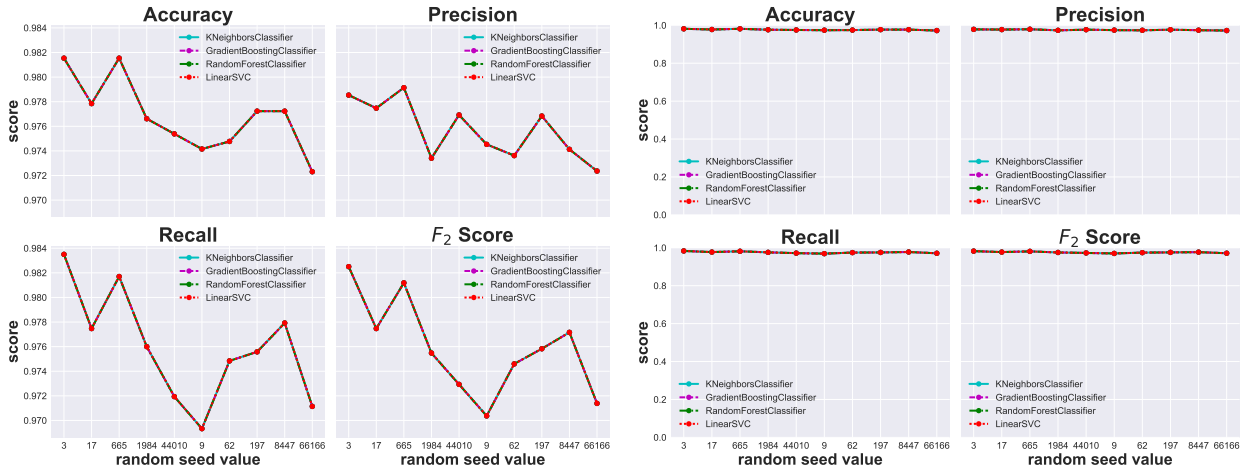
I completed my parameter tuning using Scikit-learn's *GridSearchCV()* function. I created a grid for each of the four classifiers mentioned above, tuning between four and six parameters for each. The exact grids used in this process are shown below in Figure 6. I performed the grid searches using 5-fold (stratified) cross-validation. The number of parameter combinations for each classifier was: Random Forest - 1344 (6720 total fits), Gradient Boosting - 540 (2700 total fits), k-Nearest Neighbors - 96 (480 total fits), and Linear SVC - 180 (900 total fits). The amount of time spent on this parameter tuning for each classifier was: Random Forest - 6.4 minutes, Gradient Boosting - 7.5 minutes, k-Nearest Neighbors - 3.8 minutes, and Linear SVC - 1.9 minutes. When I compare the results to those before the parameter tuning, there was no improvement in any of the cases.

4 Results

4.1 Model Evaluation and Validation

As pointed out in Table 2, when we restrict our model training to the dataset that only uses the top seven one-hot encoded features, we are using five of the original features: ODOR, GILL-SIZE, GILL-COLOR, STALK-SURFACE-ABOVE-RING, and STALK-SURFACE-BELOW-RING. The recall was only 0.004 less 0.98, which is close enough to suit me considering only five original features are used to achieve this accuracy. Hence, this is the model that I have chosen to move forward with to perform a sensitivity analysis on. Since all four of the classifiers had identical scores in this reduced feature space, I performed the sensitivity analysis on all of them. It wasn't computationally expensive. In my sensitivity analysis, I elected to fit, predict, and evaluate my chosen classifiers on a collection of different training and testing sets. I think this is the best approach. One cannot perturb categorical data with some small epsilon noise model and then re-evaluate a model's performance. If one were to modify a categorical value (or some/all of them) in a data point, it becomes an entirely different data point. In fact, it may provide no additional information as it may then be identical to another data point.

To implement the analysis, I first chose ten random integers to use as *random_state* seeds. I then passed the classifiers, the seeds, and the five feature labels into a sensitivity analysis function that I wrote: *analyze_sensitivity()*. This analysis pipeline uses the random seeds to create distinct training and testing sets. I should point out that this function does not simply perform a random shuffle on the original training and testing datasets. It takes the original one-hot encoded feature data, picks out only the requested features, and then uses *train_test_split()* to create an entirely distinct training and testing data split. This random data is then passed to *clf_eval_pipe()* in a loop to build and parse the four metrics for each of the random seeds. The results are shown in Figure 7.



(a) zoomed in

(b) zoomed out

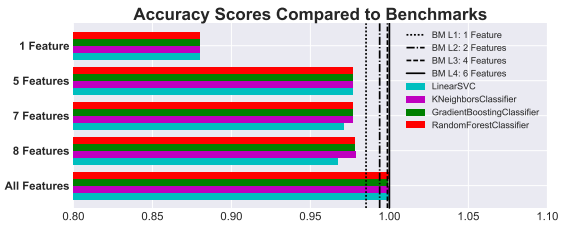
Figure 7: Sensitivity Analysis Results

The sensitivity analysis results are very promising. Figure 7a shows the results zoomed in, so that one can easily visualize that there is some variance in the metrics across the ten random seeds. Figure 7b graphs the same metrics zoomed out to the entire range of the metrics: $[0, 1]$. The first thing that we notice about these two figures is that all four of the classifiers perform identically. This is consistent with the results obtained in the model refinement process. Recall that this analysis uses the top seven one-hot encoded features. In order to quantify some of these results, consider the accuracy and precision scores. For each of the classifiers, the entire range varies by less than 1% of the maximum score. For recall and F_2, the results are nearly as good. The ranges of those two metrics vary by less than 1.5% of their maximum scores.

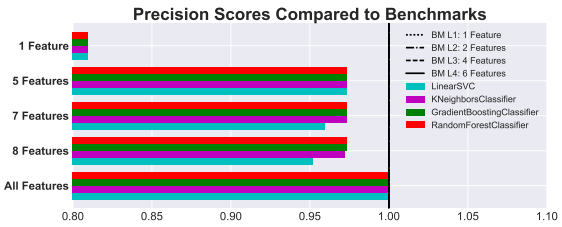
4.2 Justification

Figure 8 shows how all four of the classifiers from the sensitivity analysis compare to the benchmark. Moreover, each metric is computed for several levels of feature reduction. There is a lot of information captured in these graphs, but the most important information is contained in the bars that correspond to the "final model" that I have selected. After the sensitivity analysis and refinement processes were performed, it became evident that the Random Forest and Gradient Boosting classifiers were consistently the best performers. However, Random Forest is nearly three times faster than Gradient Boosting; during parameter tuning Random Forest took 0.057 seconds per fit *vs.* 0.167 seconds per fit for Gradient Boosting. Hence, the "final model" is the resulting Random Forest classifier after parameter tuning and trained on the top seven one-hot encoded (five original) features, henceforth referred to as "BEST CLF."

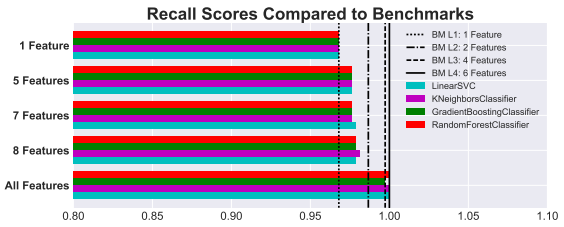
The BEST CLF does not surpass the benchmark's performance, but it is comparable. BEST CLF's recall score exceeds the level one benchmark (which only uses one original feature) and is 0.024 less than the level four benchmark which has a perfect score of 1.0. My primary goal was not to beat the benchmark, but was to design a classifier that was easily adaptable to new datasets, fast, and achieved a recall near 0.98, with an accuracy above 0.8. BEST CLF achieves this using five original features, so I would definitely claim that BEST CLF presents an acceptable solution to the problem. It is also interesting to make note of the data in Figure 8b that correspond to "1 Feature." These data match the level one benchmark which also uses only one feature. The original feature selected by my algorithm for this level of feature reduction is ODOR, the same as the single feature used by the level one benchmark (see Section 2.4).



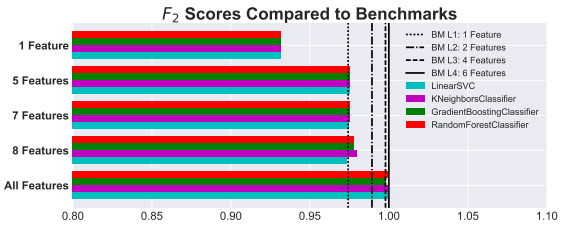
(a) Accuracy comarisons



(c) Precision comparisons



(b) Recall comparisons



(d) F_2 comparisons

Figure 8: Classifier and Benchmark Comparisons

5 Conclusion

5.1 Free-Form Visualization

In Section 2.3 I mentioned that I used a mutual information metric to reduce the number of features the models use to train the predictor. Mutual information is not the only useful metric for comparing two binary vectors. Another metric one might use is called the Matthews correlation coefficient (denoted by ϕ), which is also referred to as the Phi coefficient in some publications. Essentially, ϕ is a correlation coefficient between an observed and predicted binary classification. The values for this coefficient are always in the range $[-1,\ 1]$. A value of 1 indicates perfect prediction; a value of 0 indicates random prediction; and a value of -1 indicates perfect disagreement. Note that perfect disagreement is also very useful correlation. Scikit-learn has a built-in function to compute the Matthews correlation coefficient, and the formula may be found in many publications, e.g., Wikipedia [15]. Using the notation introduced in Section 1.3 (Metrics), the formula for ϕ is given by:

$$\phi = \frac{P_t N_t - P_f N_f}{\sqrt{(P_t + P_f)(P_t + N_f)(N_t + P_f)(N_t + N_f)}}. \tag{4}$$

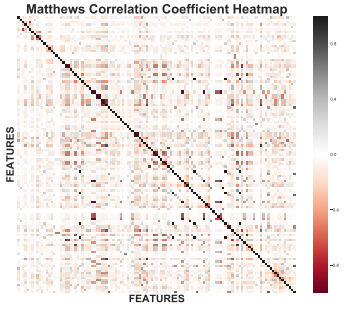
Before computing this metric for two binary vectors, we should clarify what is meant by a '*false positive*', '*true negative*', etc., because the numbers used in Equation (4) are not the ones used to compute accuracy, precision, recall, and F_2. When computing ϕ, one of the vectors is always considered '*truth*' and the other vector is the comparison, i.e., '*prediction.*' Let's consider a trivial example. Suppose you have the two binary vectors $z_1 = [0\ 1\ 1\ 1]$ and $z_2 = [0\ 0\ 1\ 1]$. If we assume z_1 is the truth vector, then $P_t = 2$, $N_t = 1$, $P_f = 0$, $N_f = 1$. We may then compute ϕ as follows:

$$\phi(z_1, z_2) = \frac{2 \cdot 1 - 0 \cdot 1}{\sqrt{(2+0)(2+1)(1+0)(1+1)}} = \frac{2-0}{\sqrt{2 \cdot 3 \cdot 1 \cdot 2}} = \frac{2}{\sqrt{12}} = \frac{\sqrt{3}}{3} \approx 0.57735.$$

We may use ϕ to compute a matrix of correlations, and then visualize that matrix as a heatmap. If we want to visualize how the features are correlated to each other, we simply iterate over the features declaring a different feature to be the '*truth*' in each iteration. One could think of this '*truth*' feature to be a pivot

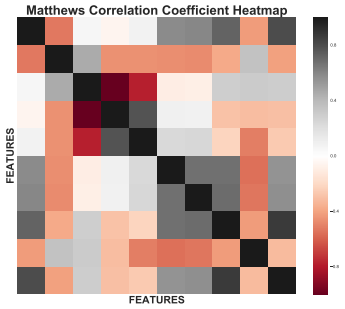
as we compute a row in the correlation matrix. In other words the first row in the matrix would be $r_0 = [\phi(z_0, z_0)\ \ \phi(z_0, z_1)\ \ \phi(z_0, z_2)\ \ \ldots\ \ \phi(z_0, z_n)]$ where n is the number of features. Two things are obvious: the matrix will be symmetric, and every value along the diagonal will be 1. We could also compute a correlation vector, where we iterate through the feature vectors but the '*truth*' (or pivot) vector is always the labels from our data. This would allow us to view how each feature is correlated to the true labels of our dataset. The results of this process are shown in Figure 9.

To interpret the feature-to-feature heatmaps, recall that high correlation indicates that we may not need both features in order to extract the information necessary to design a predictor of the true labels. Hence, in Figures 9a, 9e, 9c, and 9g, locations of dark red or dark gray indicate highly correlated features, i.e., overlapping information. In those graphs, we want to see regions (especially entire columns) that range from light red, to white, to light gray. Such rows would indicate features that we would want to keep if we were to reduce the feature space from the one producing that heatmap. In Figures 9b, 9f, 9d, and 9h, the opposite is true. We desire high correlation, i.e., dark reds and dark grays, between the features and the labels. Columns (or rows) that are very dark indicate features that are likely good predictors of the labels. As we decrease the feature space from 117 to 5, notice how these feature-to-label heatmaps get darker.



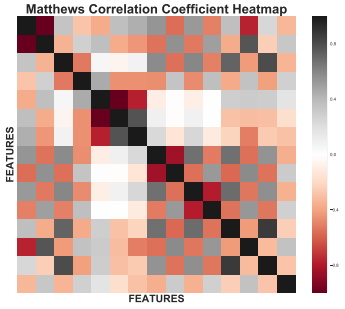
(a) Feature-to-Feature heatmap for all 117 features

(b) Feature-to-Label heatmap for all 117 features



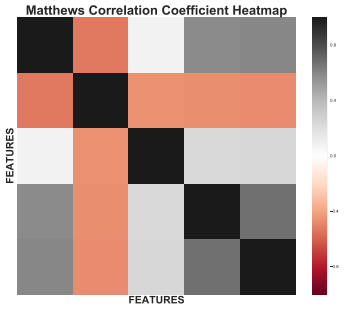
(c) Feature-to-Feature heatmap for the top 10 features

(d) Feature-to-Label heatmap for the top 10 features



(e) Feature-to-Feature heatmap for the top 15 features

(f) Feature-to-Label heatmap for the top 15 features



(g) Feature-to-Feature heatmap for the top 5 features

(h) Feature-to-Label heatmap for the top 5 features

Figure 9: Matthews Correlation Coefficient Heatmaps

5.2 Reflection

This project encompassed many aspects of a typical supervised learning project. Initial data exploration revealed the necessity for one-hot encoding and missing data replacement. Further investigation of the categorical distributions of the original 22 features revealed that most of the features had data concentrated in subsets of their possible categories, and in some cases this concentration was significant.

Then, I decided to narrow down a good classification model by examining eight classifiers. I coded an evaluation pipeline to perform all aspects of the analysis and display the results in a summary graph. I eliminated two classifiers quickly, but six of them proceeded to my refinement stage of the project. In order to make useful comparisons to the benchmark, a primary goal became feature reduction. As such, feature selection became the first aspect of the refinement stage. Fortunately, feature selection was easily implemented. I coded this capability into my analysis pipeline by way of a boolean keyword argument. After this process concluded, I was able to eliminate two of the six remaining classifiers before moving on to the next stage.

After feature selection, parameter tuning was performed. This was an especially frustrating part of the project for me. No matter how many parameters I tuned (or the number of values for each parameter), I could not get any improvement in the evaluation metrics. As an example, consider my tuning of the Gradient Boosting parameters. At one point, I performed a grid search that took 83 minutes to run with zero improvement in the metrics. It seemed that the default classifiers extracted all of the predictive information from the data. Perhaps this is more revealing of the data itself, than of the classifiers.

A sensitivity analysis was then performed on the four classifiers that I had done the parameter tuning on. Essentially, I randomly split the original data into ten distinct training and testing sets and then re-ran each one through the analysis pipeline. I then collected the metric data and graphed it. The results showed that the classifiers all yielded metric scores with less than 1.5% variance from maximum scores. Nevertheless, I was able to use all of the results to choose the classifier that was the fastest and consistently among the top performers: the default Random Forest classifier using the top seven one-hot encoded features. I then compared the results to the benchmark. It was obvious that my best classifier had not outperformed the benchmark, but it was comparable.

One of the most enjoyable aspects of the project was the free-form visualization. I have had experience producing heatmaps in the past, so I thought it would be a good idea to try and incorporate this kind of visualization to compare the features to themselves and also the features to the labels. The latter can be a good indicator of which features are likely a good predictor of the correct label. To make the visualizations a bit more exploratory, I decided to use a metric other than mutual information, which was used to perform the feature selection. I settled on a metric that is basically a correlation coefficient for binary vectors: the Matthews correlation coefficient. Overall I believe my solution meets my expectations for the problem stated in Section 1.2. I intentionally coded the project notebook so that nearly all of it could be used on a different (but similar) dataset with minimal adjustment. I think this analysis would work quite well on similar problems.

5.3 Improvement

There are several good opportunities for improvement in this project. One of them would be to obtain a more extensive dataset. As mentioned in Section 1.1, this data set only contains 8124 hypothetical samples across 23 species in Agaricus and Lepiota. But there are hundreds of species of mushrooms commonly encountered in North America alone. It would be nice to have a dataset with 250,000 to 500,000 samples across several orders of fungi. Perhaps with vastly more samples, the information embedded within the data would have sufficient structure that would require the need for significant parameter tuning, unlike the dataset that I used.

However, the main improvement that I can think of for this project is to modify the algorithmic link between the feature selection and prediction steps. I have used standard functionality within *sklearn*, but

the benchmark (which precedes the development of *sklearn*) is able to achieve perfect accuracy using only six features, whereas I am only able to achieve about 98% accuracy with five features. I think it would be worthy of research to make use of some of *sklearn*'s built-in functionality while also developing an algorithm that introduces new capability in the **feature selection** $\Longleftrightarrow$ **prediction** feedback loop. Such an investigation might be able to extract all of the predictive information out of this dataset.

References

[1] Stamets, P., *Growing Gourmet and Medicinal Mushrooms,* 3^{rd} *ed.*, Ten Speed Press, Berkeley, CA., 2000.

[2] Stamets, P. and Chilton, J.S., *The Mushroom Cultivator: A Practical Guide to Growing Mushrooms at Home*, Agarikon Press, Seattle, WA., 1983.

[3] Lincoff, G., *National Audubon Society Field Guide to North American Mushrooms*, Knopf; A Chanticleer Press edition, New York, NY., 1981.

[4] Haze, V. and Mandrake, K., *The Psilocybin Mushroom Bible: The Definitive Guide to Growing and Using Magic Mushrooms*, Green Candy Press, San Francisco, CA., 2016.

[5] Oder, T., *Wild mushrooms: What to eat, what to avoid*, retrieved from the Mother Nature Network website: https://www.mnn.com/your-home/organic-farming-gardening/stories/wild-mushrooms-what-to-eat-what-to-avoid on January 8, 2018.

[6] UCI Machine Learning Repository, *Mushroom Data Set*, retrieved from the UCI website: https://archive.ics.uci.edu/ml/datasets/Mushroom on January 8, 2018.

[7] Kaggle, *Mushroom Classification*, retrieved from the Kaggle website: https://www.kaggle.com/uciml/mushroom-classification/data on January 8, 2018.

[8] Duch, W., Adamczak, R., and Grabczewski, K., "Extraction of Logical Rules from Training Data using Backpropagation," *Proceedings of the First Online Workshop on Soft Computing*, August 1996, pp. 25-30.

[9] Howe, N. and Cardie, C., "Examining Locally Varying Weights for Nearest Neighbor Algorithms," *Proceedings of the International Conference on Case-Based Reasoning*, Providence, RI, July 1997, pp. 455-466.

[10] Li, J., Dong, G., and Ramamohanarao, K., "Instance-Based Classification by Emerging Patterns," *Proc. of the European Conference on Principles and Practice of Knowledge Discovery in Databases*, Lyon, France, Sep 2000.

[11] Kim, H. and Park, S.H., "Data Reduction in Support Vector Machines by a Kernalized Ionic Interaction Model," *Proc. of the SIAM International Conference on Data Mining*, Lake Buena Vista, FL, April 2004, pp. 507-511.

[12] Chai, X., Deng, L., Yang, Q., and Ling, C.X., "Test-Cost Sensitive Naive Bayes Classification," *Proceedings of the IEEE International Conference on Data Mining*, Brighton, UK, November 2004.

[13] Scikit-learn, *Choosing the right estimator*, retrieved from the Scikit-learn website: http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html on January 24, 2018.

[14] Scikit-learn, *Decision Trees*, retrieved from the Scikit-learn website: http://scikit-learn.org/stable/modules/tree.html#tree on January 24, 2018.

[15] Wikipedia, *Matthews correlation coefficient*, retrieved from the Wikipedia website: https://en.wikipedia.org/wiki/Matthews_correlation_coefficient on January 25, 2018.