Edibility of Mushroom Species

Ishaan Saxena isaxena@purdue.edu
Nikita Rajaneesh nrajanee@purdue.edu
Swaraj Bhaduri sbhadur@purdue.edu
Utkarsh Jain jain192@purdue.edu

November 30, 2018

1 Introduction to the Problem

1.1 Definition of the Problem

Given a dataset $\mathcal{D}$ with $n = 8124$ samples where each sample represents a mushroom with features being the observations about the characterestics of the mushrooms such as odor, color, etc., we aim to test and compare various supervised learning models for the problem of classifying each sample into either poisonous or edible. Further, we will optimize the Hyperparameters of the model which initially performs the best on the dataset.[1]

1.2 Data Description

We are given $\mathcal{D}$ with $n = 8124$ samples wherein each sample has the following 22 features (excluding the class label).

1. cap-shape
2. cap-surface
3. cap-color
4. bruises
5. odor
6. gill-attachment
7. gill-color
8. stalk-root
9. stalk-surface-above-ring
10. stalk-surface-below-ring
11. stalk-color-above-ring
12. stalk-color-below-ring
13. veil-type
14. veil-color
15. ring-type
16. spore-print-color
17. habitat
18. gill-spacing
19. gill-size
20. stalk-shape
21. ring-number
22. population

These features have been further enumerated in Appendix A.

The labels are binary, namely edible (e) and poisonous(p). Within the original dataset, there are the following number of edible and poisonous mushrooms.

```
>>> df['class'].value_counts()
e    4208
p    3916
```

[1] This dataset can be found at https://www.kaggle.com/uciml/mushroom-classification.

1.3 Encoding the Data

Note that all the features in our dataset are categorical variables. As a result, to proceed with evaluation of model performance, we must first encode these variables into numerical/binary values.

We need to deal with two kinds of categorical variables when encoding the features into numerical data. These are ordinal categorical variables and nominal categorical variables.[2] We will use different techniques to encode both of these kinds of categorical variables as they inherently represent different kinds of categorical data.

1.3.1 Nominal Categorical Variables

To encode nominal categorical variables, we will use one-hot binary features. For instance, if a feature f from a feature set $\mathcal{F}$ has k different categorical values, we can create k different binary features for each feature f of this kind. This is done because the values of each such feature do not hold any ordinal information, in that there should not be different weights for having a specific value of a specific feature.

1.3.2 Ordinal Categorical Variables

The ordinal categorical variables, on the other hand, have been encoded as numerical labels, as these informations contain valueable information about the 'scale' of a certain feature. For instance, if a feature f from a feature set $\mathcal{F}$ has k different features, it would be changed into numeric values $i \in 0, 1, ..., k-1$.

1.3.3 Encoding Process

The following code segment is used to encode the data.[3]

```
def encode(df):
    # Encode Ordinal Variables
    ordinal_columns = ['gill-spacing', 'gill-size',
            'stalk-shape', 'ring-number', 'population', 'class']
    columns = ordinal_columns[:]

    for column in columns:
        df[column] = df[column].astype('category')

        columns = df.select_dtypes(['category']).columns
        df[columns] = df[columns].apply(lambda x: x.cat.codes)

    # Encoding Nominal Variables
    columns = ordinal_columns[:]

    for column in df:
        if column not in columns:
            dummies = pd.get_dummies(df.pop(column))
            column_names = [column + "_" + x for x in dummies.columns]
            dummies.columns = column_names
            df = df.join(dummies)

    return df
```

[2]Information on which features are which kind of categorical variables can be found in Appendix A.
[3]This can be found in the data.py file.

We can now see how our encoded dataset $\mathcal{D}$ is shaped, and then proceed further in our task to classify data.

```
>>> X, y = data.load() # data.load() calls data.encode()
>>> X.shape
(8124, 107)
>>> y.shape
(8124,)
```

2 Feature Selection

2.1 The Need for Feature Selection

Feature selection is a useful technique when we want to better understand the input feature set. Having fewer features also leads to a more interpretable model, and is definitely helpful in our case, since we have just created over 100 features from which we want to derive a model to solve the classification problem. As a result, feature selection techniques can help us get a better idea of which features matter more or less while classifying our model.

It is also important to note that feature selection helps us have a control over the complexity of the model. This is a very important aspect, as a less complex model is more useful for generalization.

2.2 Feature Selection Technique

We are using Greedy Subset Selection for Feature Selection. We will talk more about the selection technique and its implementation in the final report.

2.3 Greedy Subset Selection Results

The following are the 46 best features and their weights obtained from Greedy Subset Selection:

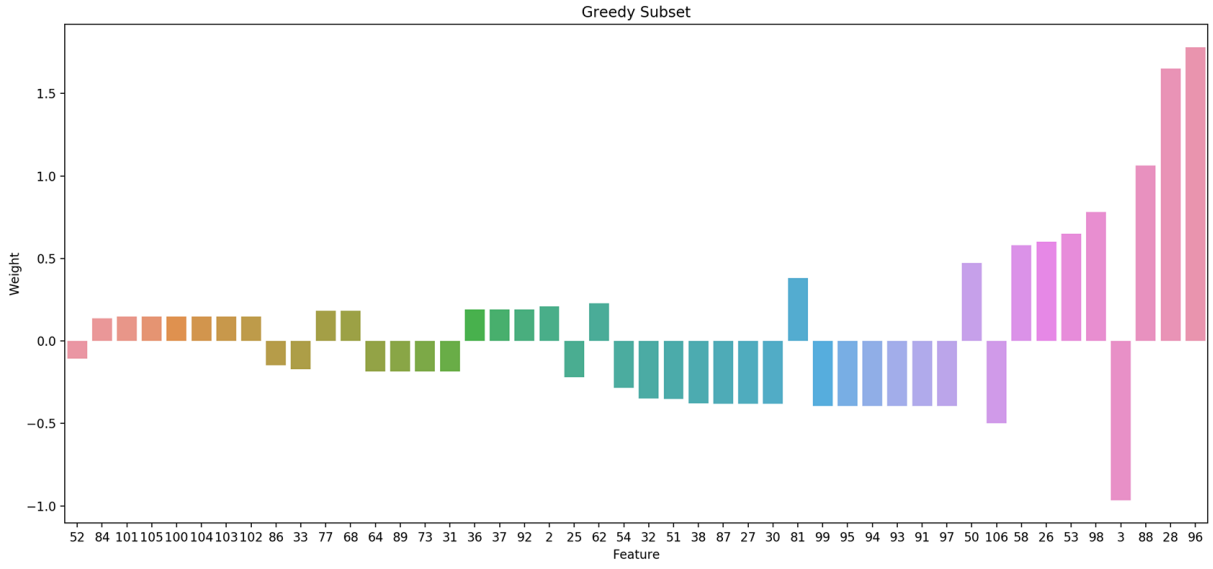

Figure 1: Greedy Subset Selection - 46 features with highest weights.

We will talk more about reduction of features, how that impacts our problem, and about what these specific features are in the final report.

3 Machine Learning Models

In order to solve the classification problem, we will test the following machine learning algorithms with $\{2, 5, 10\}$-Fold Cross Validation and different metrics, including Hypothesis Testing, ROC Curves, etc. to compare the results:

1. Perceptron
2. Support Vector Machine
3. Logistic Regression
4. Adaboost

3.1 Cross-Validation Technique: k-Fold CV

We use cross-validation to evaluate the chosen predictive model by paritioning it into smaller train and test datasets wherein we train the model on the training set and evaluate it on the testing set. By using better cross-validation techniques, we can get a better understanding of how well our model will generalize, and whether we might be over or underfitting.

Here, we are using k-Fold Cross Validation on 2, 5, and 10 folds. In k-Fold CV, we split the dataset into k different folds. For each i^{th} fold, where $i = 1, 2, ..., k$, we train on the remaining $k - 1$ folds and evaluate on the single remaining fold. As a result, we get k different measures of the metrics which we are using for each model we evaluate using the cross-validation technique. These can be aggregated into a single metric as follows:

$$\hat{\mu} = \frac{1}{k} \sum_{1 \leq i \leq k} M_i, \text{ and } \hat{\sigma}^2 = \frac{1}{k} \sum_{1 \leq i \leq k} (M_i - \hat{\mu})^2$$

where M_i is the measure for the i^{th} fold (here, it is accuracy). This aggregation may be used in a statistical hypothesis test to compare the performances of two different models.

The following is our implementation of k-fold cross validation:[4]

```
# For each fold
for i in range(k):
    # Find sets S and T
    lower = np.floor(float(n) * i / k).astype('int64')
    upper = np.floor(float(n) * (i + 1) / k - 1).astype('int64')
    T = range(lower, upper + 1)
    S = list(range(0, lower)) + list(range(upper + 1, n))

    # Get training data
    X_train = X[S, :]
    y_train = y[S]
```

[4]This can be found in the kfoldcv.py file.

```
        # Fit model
        m = model(**kwargs)
        m.fit(X_train, y_train)

        # Check prediction accuracy
        y_pred = m.predict(X[T])

        # Update Z values based on accuracy score
        Z[i] = metrics.accuracy_score(y[T], y_pred)

    # Return k-Fold results
    return Z
```

To see how this we utilize this script to get results for specific models, please see Appendix B.

3.2 Model k-Fold CV Results

3.2.1 Perceptron

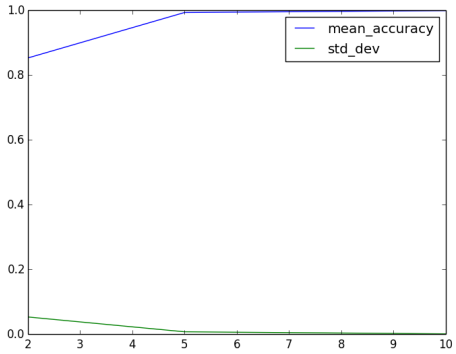

Figure 2: Perceptron accuracy in 2,3,5-Fold CV

We get the following results for a Perceptron when we run kfoldcv_graph.py:

```
Model: Perceptron
    2-fold: (mu=0.853, sigma=0.054)
    5-fold: (mu=0.994, sigma=0.008)
    10-fold: (mu=1.000, sigma=0.001)
    Time: 9.351s
```

3.2.2 SVM

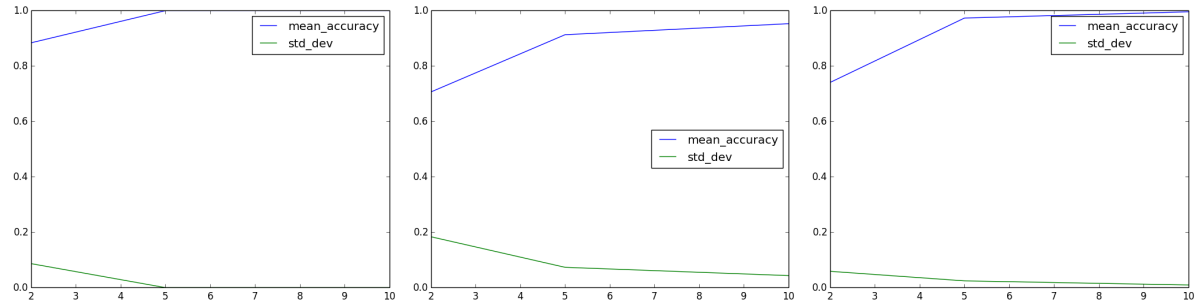

Figure 3: SVM accuracy in 2,3,5-Fold CV. From left to right, we are using linear, polynominal, and rbf kernels.

We get the following results for SVMs when we run kfoldcv_graph.py:

```
Model: SVM-linear
    2-fold: (mu=0.884, sigma=0.087)
    5-fold: (mu=1.000, sigma=0.000)
    10-fold: (mu=1.000, sigma=0.000)
    Time: 5.442s
Model: SVM-poly
    2-fold: (mu=0.707, sigma=0.184)
    5-fold: (mu=0.913, sigma=0.073)
    10-fold: (mu=0.953, sigma=0.043)
    Time: 36.912s
Model: SVM-rbf
    2-fold: (mu=0.741, sigma=0.059)
    5-fold: (mu=0.973, sigma=0.024)
    10-fold: (mu=0.996, sigma=0.010)
    Time: 13.458s
```

3.2.3 Logistic Regression

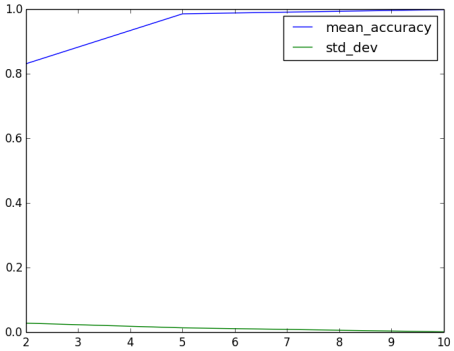

Figure 4: Logistic Regression accuracy in 2,3,5-Fold CV.

We get the following results for Logistic Regression when we run kfoldcv_graph.py:

```
Model: LR
    2-fold: (mu=0.832, sigma=0.029)
    5-fold: (mu=0.986, sigma=0.014)
    10-fold: (mu=0.999, sigma=0.002)
    Time: 0.754s
```

3.2.4 Adaboost

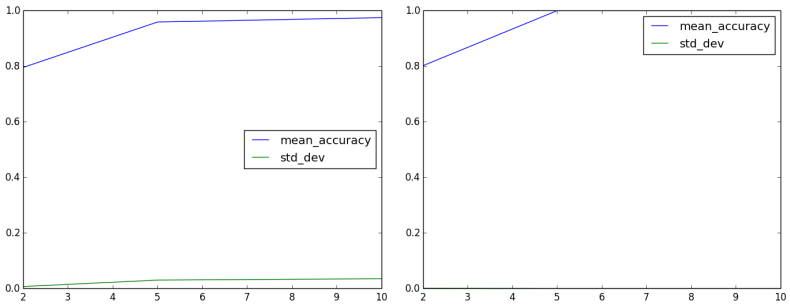

Figure 5: Adaboost accuracy in 2,3,5-Fold CV. From left to right, we have 10 and 50 $n_estimators$

We get the following results for Adaboost when we run kfoldcv_graph.py:

```
Model: Adaboost-n_estimators 10
    2-fold: (mu=0.796, sigma=0.007)
    5-fold: (mu=0.959, sigma=0.031)
    10-fold: (mu=0.975, sigma=0.035)
    Time: 1.437s
Model: Adaboost-n_estimators 50
    2-fold: (mu=0.802, sigma=0.001)
    5-fold: (mu=1.000, sigma=0.000)
    10-fold: (mu=1.000, sigma=0.000)
    Time: 6.402s
```

Appendix A: Data Description

Classes: edible=e, poisonous=p (y-values)

Size of dataset (before encoding): $(n = 8124, d = 22)$
Size of dataset (after encoding): $(n = 8124, d = 107)$
Attribute Information and Encoding:

1. Nominal Categorical Variables:
   These variables will be encoded as binary one-hot features. As a result, each feature in this category would be replace by the k features in the encoded dataset if the feature has k possible values. These featues include:
   i. **cap-shape**: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
   ii. **cap-surface**: fibrous=f, grooves=g, scaly=y, smooth=s
   iii. **cap-color**: brown=n, buff=b, cinnamon=c, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y
   iv. **bruises**: bruises=t, no=f
   v. **odor**: almond=a, anise=l, creosote=c, fishy=y, foul=f, musty=m, none=n, pungent=p, spicy=s
   vi. **gill-attachment**: attached=a, descending=d, free=f, notched=n
   vii. **gill-color**: black=k, brown=n, buff=b, chocolate=h, gray=g, green=r, orange=o, pink=p, purple=u, red=e, white=w, yellow=y
   viii. **stalk-root**: bulbous=b, club=c, cup=u, equal=e, rhizomorphs=z, rooted=r, missing=?
   ix. **stalk-surface-above-ring**: fibrous=f, scaly=y, silky=k, smooth=s
   x. **stalk-surface-below-ring**: fibrous=f, scaly=y, silky=k, smooth=s
   xi. **stalk-color-above-ring**: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
   xii. **stalk-color-below-ring**: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
   xiii. **veil-type**: partial=p, universal=u
   xiv. **veil-color**: brown=n, orange=o, white=w, yellow=y
   xv. **ring-type**: cobwebby=c, evanescent=e, flaring=f, large=l, none=n, pendant=p, sheathing=s, zone=z
   xvi. **spore-print-color**: black=k, brown=n, buff=b, chocolate=h, green=r, orange=o, purple=u, white=w, yellow=y
   xvii. **habitat**: grasses=g, leaves=l, meadows=m, paths=p, urban=u, waste=w, woods=d
2. Ordinal Categorical Variables:
   These variables will be encoded in place by encoding labels, as the data here has ordinal meaning to it. These variables include:
   i. **gill-spacing**: close=c→0, crowded=w→1, distant=d→2
   ii. **gill-size**: broad=b→0, narrow=n→1
   iii. **stalk-shape**: enlarging=e→0, tapering=t→1
   iv. **ring-number**: none=n→0, one=o→1, two=t→2
   v. **population**: abundant=a→0, clustered=c→1, numerous=n→2, scattered=s→3, several=v→4, solitary=y→5

Appendix B: k-Fold CV Graphs

The following code an excerpt from the file kfoldcv_graph.py. This file uses the kfoldcv.py module to get metrics for multiple machine learning models and saves plots of accuracy vs number of folds for each of the models.

```
# Import models. For example:
from sklearn.svm import SVC

models = []
# Add models to list. For example:
models.append(("SVM-linear", SVC, {'kernel':'linear'}))
models.append(("SVM-poly", SVC, {'kernel':'poly'}))

# For each algorithm
for name, model, kwargs in models:
    # Folds
    folds = [2, 5, 10]
    # Mean accuracy and std dev
    m = []
    s = []
    # For each fold size
    for fold_size in folds:
        Z = kfoldcv.run(X, y, model, fold_size, **kwargs)
        m.append(np.mean(Z))
        s.append(np.std(Z))
    # Plot accuracy vs. fold size
    df = pd.DataFrame({
        'mean_accuracy': m,
        'std_dev': s
    }, index = folds)
    lines = df.plot.line()
    filename = project.results + "model_accuracy_vs_folds_" + name + ".png"
    plt.savefig(filename, bbox_inches='tight')
```

This is how the output looks when this script is run:

```
    Model:      SVM-linear
        2-fold: (mu=0.884, sigma=0.087)
        5-fold: (mu=1.000, sigma=0.000)
        10-fold: (mu=1.000, sigma=0.000)
        Time: 4.987s
    Model:      SVM-poly
        2-fold: (mu=0.707, sigma=0.184)
        5-fold: (mu=0.913, sigma=0.073)
        10-fold: (mu=0.953, sigma=0.043)
        Time: 33.005s
```

This file also saves graphs like in section 3 for different models.

Appendix C: List of Scripts

The following scripts have been included with the preliminary report in the zip file:

1. data.py - Interface to load encoded data.
2. project.py - Contains project config.
3. kfoldcv.py - Conduct kfoldcv.
4. kfoldcv_graph.py - Conduct kfoldcv for multiple models and multiple values of k and save the results.
5. greedySubset.py - Conduct Greedy Subset Selection.
6. featureSelection.py - Graph the results for Greedy Subset Selection.