Classifying Mushrooms for Edibility by Using Machine Learning Algorithms

Kutay Erkan*

Bahçeşehir University, Big Data Analytics and Management. 34347 Beşiktaş, İstanbul.

Abstract

Mushrooms are an important part of many people's diet. But this doesn't change the fact that some of them are harmful for a human's digestive system. A small percentage of mushrooms can even kill a human. This is why classifying which mushrooms are edible is not a redundant task. In this study, a dataset from UC Irvine Machine Learning Repository with 8123 mushroom samples was used. Python language was used for coding, and sklearn library was used extensively. After preprocessing and encoding; Gaussian Naïve Bayes, Logistic Regression, and Decision Tree algorithms were used to classify mushrooms. Finally, performance metrics for these algorithms are shown and discussed briefly.

Keywords: mushroom; classification; naïve bayes; logistic regression; decision tree; python

Mushroom Classification

Dataset for this classification task was imported from UC Irvine Machine Learning Repository. Several steps that were followed during the analysis are as follows:

- Preprocessing the data
- Applying machine learning algorithms to data
- Comparing performance of different ML algorithms

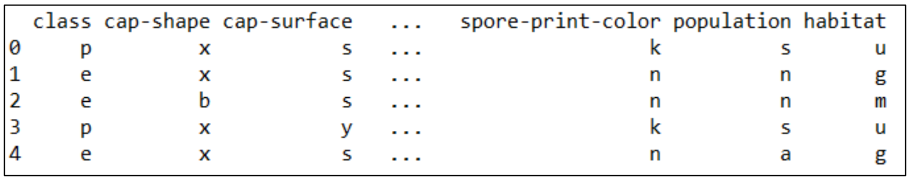
Each step of the analysis will be discussed separately.

* Corresponding author. Tel.: +90-539-269-7980;
E-mail address: kutay.erkan@bahcesehir.edu.tr

1.1. Preprocessing

After reading the data into a Pandas dataframe, we first check if there are any missing data. It would be unlikely considering it is a curated dataset, but it is still good practice. We see that we don't have any missing data.

Afterwards we take a look at our dataset (many of the columns are truncated). Further information about the features can be seen at Appendix A.



```
  class cap-shape cap-surface    ...    spore-print-color population habitat
0     p         x           s    ...                    k          s       u
1     e         x           s    ...                    n          n       g
2     e         b           s    ...                    n          n       m
3     p         x           y    ...                    k          s       u
4     e         x           s    ...                    n          a       g
```

Fig. 1. The dataset after importing

Upon further inspection of the dataset, the feature called "veil-type" has only one unique value ("p", which stands for partial). We wouldn't possibly gain any information from this feature, which means we also wouldn't use it in a classification task. That feature is removed.

We also see that all of our features are categorical, so to be able to use many of the machine learning algorithms directly, we use label encoding.

With encoding, we are now able to see a histogram of our features and label(class):

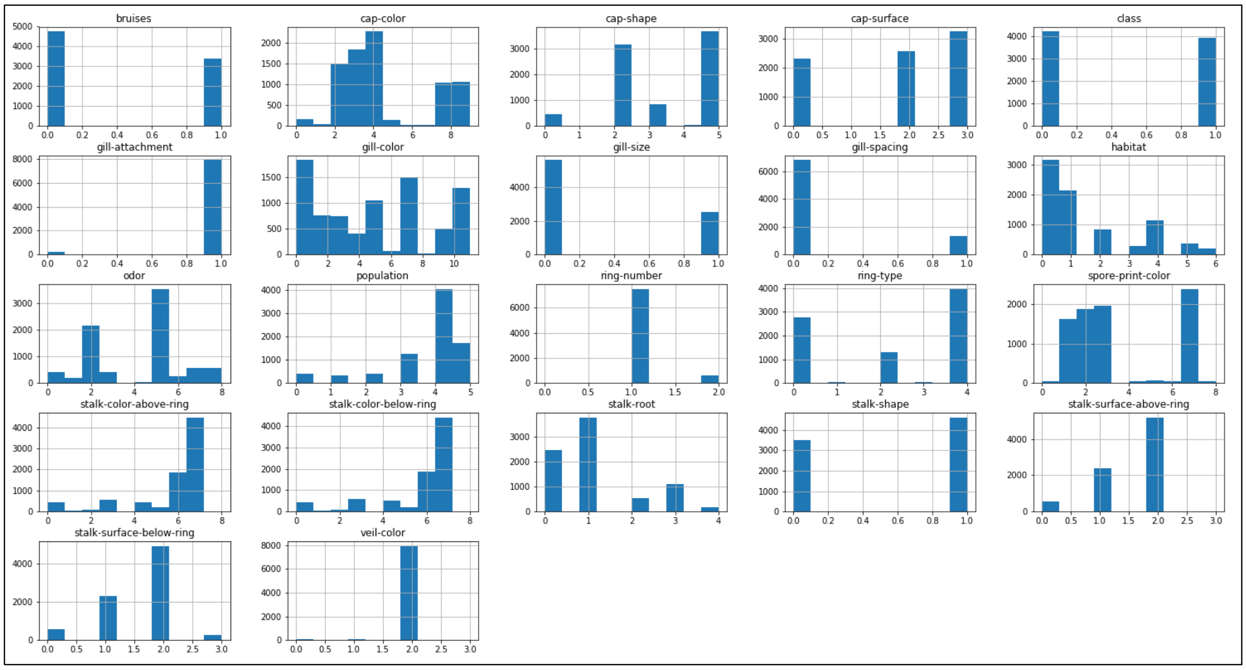


Fig. 2. Histogram of features and label

We see that there are some features with very low variance, in other words features that are almost with one unique value (like veil-type). But these features still could be really useful if they separate the class well.

We also look at the correlation matrix of our features and label:

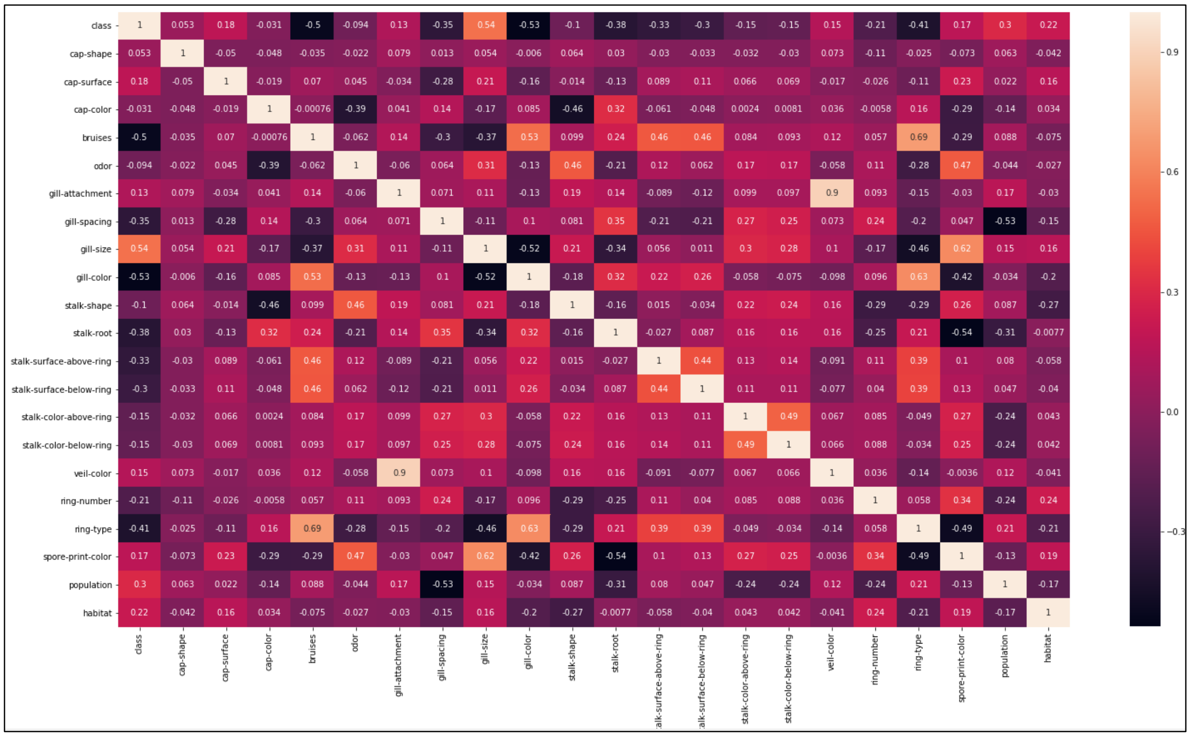


Fig. 3. Correlation matrix of features and label

"Veil-color" and "gill-attachment" have a 0.9 correlation coefficient between them, but we know this is due to the fact that they each have very low intra-class variance. We will look at feature importances, but it is good practice to keep data unless there is a solid reason to remove, so we keep them for now.

Finally, we check our label to see if our dataset is balanced. If it is not, we might need some measures while separating our dataset to training data, validation data, and test data; and also it must be considered while applying algorithms. We see that dataset is very balanced (48%/52%), which is good for us. We separate the dataset as follows:

- Training set (50%)
- Validation set (25%)
- Test set (25%)

1.2. Analysis

First step of the analysis is to check feature importance using a random forest classifier. Features with the least importance are “cap-color”, “veil-color”, “gill-attachment”, and “cap-shape”. These features are always never used in a random forest classifier. Keeping also in mind that “veil-color” and “gill-attachment” have very little intra-class variance, we can remove these features to avoid overfitting and increase performance.

After splitting our reduced again, we start applying ML algorithms to our dataset. First algorithm is Gaussian Naïve Bayes. Naïve Bayes is a very fast algorithm computation-wise, and it gives good results even though its condition (independent features) is seldom completely satisfied. It also doesn’t have hyperparameters that require tuning, so we apply it to both training and validation sets.

Gaussian NB Performance on Training Data:

- Accuracy: 0.900
- Recall: 0.848
- Precision: 0.938
- f1: 0.891
- Confusion Matrix:

```
[[1995  109]
 [ 298 1660]]
```

Gaussian NB Performance on Validation Data:

- Accuracy: 0.895
- Recall: 0.829
- Precision: 0.946
- f1: 0.884
- Confusion Matrix:

```
[[1006   46]
 [ 167  812]]
```

Gaussian Naïve Bayes returns good results, it especially does well in precision. But we still classify 17% of poisonous mushrooms incorrectly on the validation dataset.

Second algorithm will be Logistic Regression. Performance metrics are as follows:

Logistic Regression Performance on Training Data:

- Accuracy: 0.936
- Recall: 0.918
- Precision: 0.948
- f1: 0.933
- Confusion Matrix:

```
[[2005   99]
 [ 160 1798]]
```

Results are better compared to Naïve Bayes, but Logistic Regression allows us to tune its hyperparameters. We use the validation set for this task. The grid is {"C":[0.01,0.1,1.0,10.0,100.0,1000.0], "penalty":["l1","l2"]}

Selected hyperparameters are {'C': 1000.0, 'penalty': 'l2'}. We use these hyperparameters to predict for the labels in validation data.

Logistic Regression Performance on Validation Data:

- Accuracy: 0.923
- Recall: 0.893
- Precision: 0.945
- f1: 0.918
- Confusion Matrix:

```
[[1001   51]
 [ 105  874]]
```

The results are worse than what we have in training data, but that is expected as we trained our model with the data from training dataset. Afterwards we tuned our hyperparameters using validation dataset.

Final algorithm we use is Decision Tree. Decision Tree algorithms are prevalent in Machine Learning domain, and it is easy to see why: They are relatively fast (or can be adjusted to run fast), and also very easy to understand even if one doesn't have Machine Learning experience.

The performance of the Decision Tree with default hyperparameters are:

Decision Tree Performance on Training Data:

- Accuracy: 1.0
- Recall: 1.0
- Precision: 1.0
- Confusion Matrix:

```
[[2104    0]
 [   0 1958]]
```

This is interesting. Default Decision Tree algorithm completely predicts the label of every mushroom in the training dataset. Two explanations come to mind: Either our Decision Tree algorithm overfitted the training data and learned not only necessary information but also noise, or we don't have noise in the dataset and what the model learned from the training dataset is enough to predict class of all mushrooms. Usually the first explanation is much more likely, but second is also possible. One way to understand this is to check our model against the validation dataset. Since the decision tree didn't learn from validation dataset, it should perform considerably worse if it overfitted the training dataset. One pitfall here is separating the datasets correctly, if we didn't do it correctly our model also learns from the validation dataset – and we might not even notice.

Decision Tree Performance on Validation Data:

- Accuracy: 1.0
- Recall: 1.0
- Precision: 1.0
- Confusion Matrix:

```
[[1052    0]
 [   0  979]]
```

For this case, second explanation was true. Decision Tree algorithm also predicts the classes of every mushroom in the validation dataset.

1.3. Comparing the Performance of Naïve Bayes, Logistic Regression, and Decision Tree

We had separated 25% of our dataset as the "test set". We use this dataset for comparing three Machine Learning algorithms.

Gaussian NB Performance on Test Data:

- Accuracy: 0.905
- Recall: 0.856
- Precision: 0.943
- f1: 0.897
- Confusion Matrix:

```
[[1001   51]
 [ 141  838]]
```

Logistic Regression Performance on Test Data:

- Accuracy: 0.939
- Recall: 0.930
- Precision: 0.944
- f1: 0.937
- Confusion Matrix:

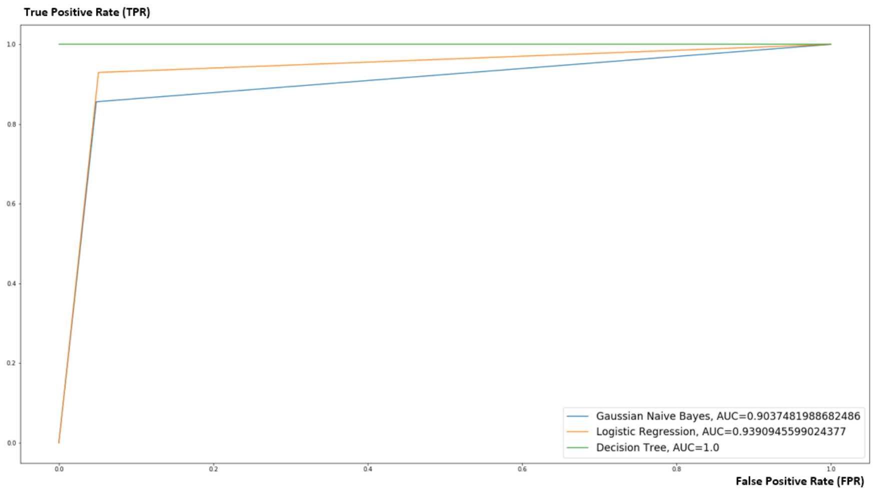
```
[[998  54]
 [ 69 910]]
```

Decision Tree Performance on Test Data:

- Accuracy: 1.000
- Recall: 1.000
- Precision: 1.000
- f1: 1.000
- Confusion Matrix:

```
[[1052    0]
 [   0  979]]
```

We see that Decision Tree is still better than other algorithms. Which other algorithm is the 2nd best? While comparing different Machine Learning there is not one performance metric that is "the best", because we have different use-cases. For classifying mushrooms, it could be said that classifying a poisonous mushroom as "edible" is much worse than not being able to eat some delicious mushrooms because we classified them as poisonous. This means for this situation, recall is more important than precision. In other words, after Decision Tree, Logistic Regression is the 2nd best algorithm, but Naïve Bayes also isn't "bad" as it has an f1 score of ~90%. We can see that also in the ROC Curve, as this order is the order they are closest to upper-left corner.



References

Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

Appendix A. Feature Information

- class: edible=e, poisonous=p
- cap-shape: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
- cap-surface: fibrous=f, grooves=g, scaly=y, smooth=s
- cap-color: brown=n, buff=b, cinnamon=c, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y
- bruises?: bruises=t, no=f
- odor: almond=a, anise=l, creosote=c, fishy=y, foul=f, musty=m, none=n, pungent=p, spicy=s
- gill-attachment: attached=a, descending=d, free=f, notched=n
- gill-spacing: close=c, crowded=w, distant=d
- gill-size: broad=b, narrow=n
- 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
- stalk-shape: enlarging=e, tapering=t
- stalk-root: bulbous=b, club=c, cup=u, equal=e, rhizomorphs=z, rooted=r, missing=?
- stalk-surface-above-ring: fibrous=f, scaly=y, silky=k, smooth=s
- stalk-surface-below-ring: fibrous=f, scaly=y, silky=k, smooth=s
- stalk-color-above-ring: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
- stalk-color-below-ring: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
- veil-type: partial=p, universal=u
- veil-color: brown=n, orange=o, white=w, yellow=y
- ring-number: none=n, one=o, two=t
- ring-type: cobwebby=c, evanescent=e, flaring=f, large=l, none=n, pendant=p, sheathing=s, zone=z
- spore-print-color: black=k, brown=n, buff=b, chocolate=h, green=r, orange=o, purple=u, white=w, yellow=y
- population: abundant=a, clustered=c, numerous=n, scattered=s, several=v, solitary=y
- habitat: grasses=g, leaves=l, meadows=m, paths=p, urban=u, waste=w, woods=d