Mushroom Classification by Toxicity

June 20, 2022

Karime Ochoa Jacinto   Anton Pashkov

***Abstract.* This article describes a machine learning project to predict the toxicity of hypothetical mushroom samples based on their physical properties using Naive Bayes and Decision Trees.**

1 Introduction

Collecting mushrooms (shrooming) is a popular hobby for many people; however, it is a rather dangerous activity even for the most experienced collectors because of the potential toxicity coupled with nature's mechanisms that cause that non-venomous species emulate venomous features as a protection mechanism, a phenomenon known as mimicry. As such, this project seeks to construct a reliable prediction model of mushrooms' toxicity.

2 Materials and Methods

The model was implemented in the Python programming language (version 3.8), as it provides all the necessary libraries for data analysis and machine learning: Pandas, Numpy, Matplotlib, and Scikit-learn. Prior to the final version of the model, several algorithms and parameters were tested, each of which are documented in this article. The mushroom dataset used in this project was downloaded from Kaggle (https://www.kaggle.com/datasets/uciml/mushroom-classification). It contains information on the color and shape of different mushrooms' caps and gills, as well as their bruises and odors. Each mushroom is classified as poisonous (p) or edible (e).

3 Experiments and Results

3.1 Importing Data

The first step was to import the dataset as a Pandas DataFrame (**Table 1**). The import required no special parameters.

[1]:
```
import pandas as pd
data = pd.read_csv("mushrooms.csv")
```

	class	cap-shape	cap-surface	cap-color	bruises
0	p	x	s	n	t
1	e	x	s	y	t
2	e	b	s	w	t
3	p	x	y	w	t
4	e	x	s	g	f

Table 1*. A glimpse into the dataset, showing the labels and some of the features available. In total, the table contains 23 columns (including the class labels) and 8124 rows.*

The dataset required no further adjustments, as it contains no missing values or non-sense information. Nevertheless, in order to apply Scikit-learn's machine learning algorithms, the labels had to be encoded to numbers (**Table 2**).

[2]:
```
class_names = data["class"].unique()

from sklearn.preprocessing import LabelEncoder
data = data.apply(LabelEncoder().fit_transform)
class_names
```

[2]:
```
array(['p', 'e'], dtype=object)
```

	class	cap-shape	cap-surface	cap-color	bruises
0	1	5	2	4	1
1	0	5	2	9	1
2	0	0	2	w	1
3	1	5	3	8	1
4	0	5	2	3	0

Table 2*. The same section as Table 1, after apply label encoding; notice how the "poisonous" labels where changed to 1 and the "non-poisonous" (edible) ones to 0.*

The dataset was then divided into a series of class labels and the corresponding features associated to each instance, an obligatory requirement for Scikit-learn's algorithms.

[3]:
```
classes = data["class"]
data.drop("class", axis=1, inplace=True)
```

Finally, the data was split into training and test sets, with each set containing two thirds and one third of the data, respectively. A random state of zero was used to guarantee reproducibility.

[4]:
```
from sklearn.model_selection import train_test_split
split = train_test_split(data, classes, test_size=1/3, random_state=0)
x_train, x_test, y_train, y_test = split
```

3.2 KN-Neighbor

The first machine learning model chosen was k-nearest neighbor because we thougt that possibly for the mimicry described in the introduction this would not obtain a fulfilling score. For this model we set k = 90 as it is a good aproximation to the square of the total number of instances.

```
[5]: from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import classification_report
     from sklearn.metrics import f1_score
     import numpy as np

     neighbors = int(np.sqrt(len(classes)))
     knn = KNeighborsClassifier(n_neighbors=neighbors)
     knn.fit(x_train, y_train)
     pred = knn.predict(x_test)
     print(classification_report(y_test, pred))

     score = f1_score(y_test, pred)
     print(score)
```

```
              precision    recall  f1-score   support

           0       0.93      0.99      0.96      1420
           1       0.99      0.92      0.95      1288

    accuracy                           0.96      2708
   macro avg       0.96      0.95      0.96      2708
weighted avg       0.96      0.96      0.96      2708

0.9524576954069299
```

3.3 Decision Trees

The second machine learning model applied were decision trees. The huge benefit of using trees is that they provide a visual output, useful for non-technical people interested in the data. Different parameters were tested in order to find the most appropiate configuration for the given data. Each setup was evaluated through five-fold cross-validation using F_1 as the scoring metric, and calculating its mean. Just like with the splitting, a random state of zero was employed.

```
[6]: from sklearn.tree import DecisionTreeClassifier as dtc
     from sklearn.model_selection import cross_val_score

     criteria = ["entropy", "gini"]
     depths = range(2, 10)
     results = pd.DataFrame(
         columns=["criterion", "depth", "score"],
         index=range(len(criteria) * len(depths))
     )
```

```
[7]: i = 0
     for c in criteria:
         for d in depths:
             model = dtc(random_state=0, criterion=c, max_depth=d)
             scores = cross_val_score(model, x_train, y_train, scoring="f1")
             results.iloc[i] = [c, d, scores.mean()]
             i += 1
```

	criterion	depth	score
5	entropy	7	1
6	entropy	8	1
7	entropy	9	1
14	gini	8	1
15	gini	9	1

Table 3. *Several configurations create a decision tree with a F_1 score of 1.*

Because there were many configurations producing great results (**Table 3**), each of the models was applied on the test data to compare their performance.

```
[8]: from sklearn.metrics import f1_score

     final_scores = results[results.score == 1]

     for i in final_scores.index:
         c = final_scores.loc[i]["criterion"]
         d = final_scores.loc[i]["depth"]
         model = dtc(random_state=0, criterion=c, max_depth=d)
         model.fit(x_train, y_train)
         y_pred = model.predict(x_test)
         score = f1_score(y_test, y_pred)
         final_scores.loc[i]["score"] = score
```

After making the predictions, the same values as in **Table 3** were obtained. One could suppose that any of the models is adequate; however, building longer trees might bias the predictions due to overfitting. As such, the chosen model uses entropy as the criterion of split quality and a depth of seven. A graphical visualization of this tree can be seen in **Figure 1**.

```
[9]: import matplotlib.pyplot as plt
     from sklearn.tree import plot_tree

     model = dtc(random_state=0, criterion="entropy", max_depth=7)
     model.fit(x_train, y_train);
```

[10]:
```python
fig, ax = plt.subplots(figsize=(21, 15))
plot_tree(
    model,
    class_names=["poisonous", "edible"],
    feature_names=data.columns,
    filled=True,
    fontsize=10,
    ax=ax
);
```

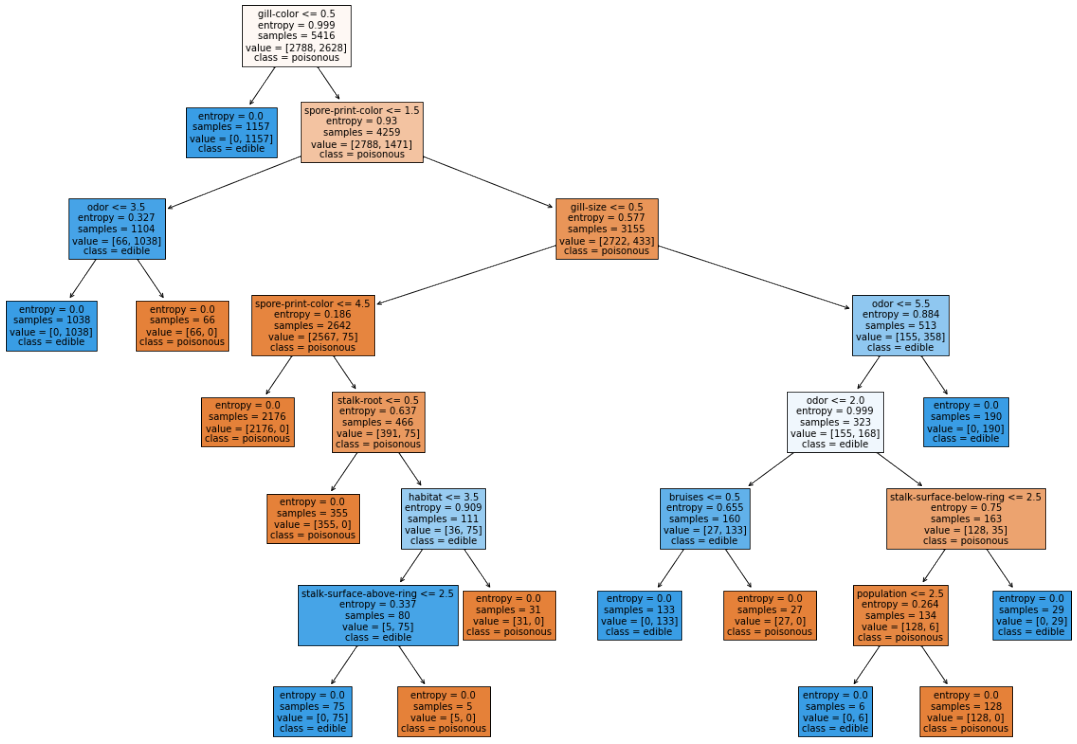
gill-color <= 0.5 entropy = 0.999 samples = 5416 value = [2788, 2628] class = poisonous
entropy = 0.0 samples = 1157 value = [0, 1157] class = edible
spore-print-color <= 1.5 entropy = 0.93 samples = 4259 value = [2788, 1471] class = poisonous
odor <= 3.5 entropy = 0.327 samples = 1104 value = [66, 1038] class = edible
gill-size <= 0.5 entropy = 0.577 samples = 3155 value = [2722, 433] class = poisonous
entropy = 0.0 samples = 1038 value = [0, 1038] class = edible
entropy = 0.0 samples = 66 value = [66, 0] class = poisonous
spore-print-color <= 4.5 entropy = 0.186 samples = 2642 value = [2567, 75] class = poisonous
odor <= 5.5 entropy = 0.884 samples = 513 value = [155, 358] class = edible
entropy = 0.0 samples = 2176 value = [2176, 0] class = poisonous
stalk-root <= 0.5 entropy = 0.637 samples = 466 value = [391, 75] class = poisonous
odor <= 2.0 entropy = 0.999 samples = 323 value = [155, 168] class = edible
entropy = 0.0 samples = 190 value = [0, 190] class = edible
entropy = 0.0 samples = 355 value = [355, 0] class = poisonous
habitat <= 3.5 entropy = 0.909 samples = 111 value = [36, 75] class = edible
bruises <= 0.5 entropy = 0.655 samples = 160 value = [27, 133] class = edible
stalk-surface-below-ring <= 2.5 entropy = 0.75 samples = 163 value = [128, 35] class = poisonous
stalk-surface-above-ring <= 2.5 entropy = 0.337 samples = 80 value = [5, 75] class = edible
entropy = 0.0 samples = 31 value = [31, 0] class = poisonous
entropy = 0.0 samples = 133 value = [0, 133] class = edible
entropy = 0.0 samples = 27 value = [27, 0] class = poisonous
population <= 2.5 entropy = 0.264 samples = 134 value = [128, 6] class = poisonous
entropy = 0.0 samples = 29 value = [0, 29] class = edible
entropy = 0.0 samples = 75 value = [0, 75] class = edible
entropy = 0.0 samples = 5 value = [5, 0] class = poisonous
entropy = 0.0 samples = 6 value = [0, 6] class = edible
entropy = 0.0 samples = 128 value = [128, 0] class = poisonous

Figure 1. *The final decision tree constructed for mushroom toxicity prediction.*

4 Conclusions

The results of this project might take the readers to the wrong direction regarding the poisonous nature of mushrooms. As mentioned in the introduction, many biological creatures employ mimicry for survival purposes, meaning that it is possible to stumble upon poisonous fungi that look like edible, and viceversa. However, at least for the data presented in the dataset, the results are extremely satisfying, and might be used as a “general rule of thumb” when classifying mushrooms.

5 References

- UCI Machine Learning (2017). Mushroom Classification: Safe to eat or deadly poison? Retrieved from https://www.kaggle.com/datasets/uciml/mushroom-classification