Mushrooms

Dataset - https://www.kaggle.com/datasets/uciml/mushroom-classification/

Authors - Shawn Lau, Evan Millar

Introduction -

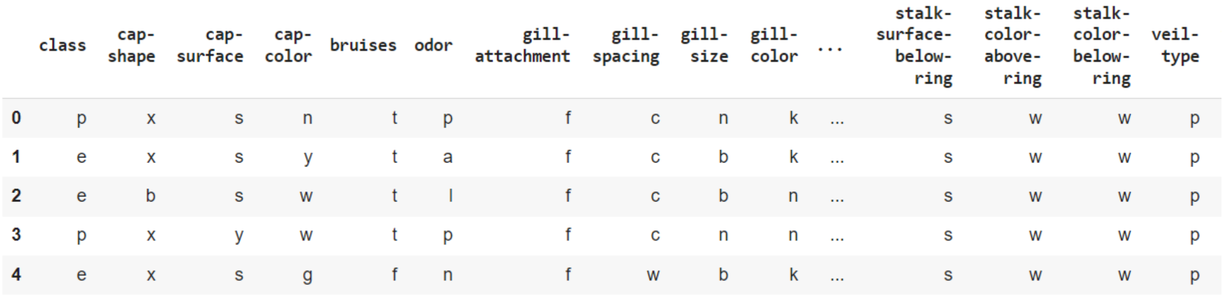
We wanted to know if it is possible to determine whether a mushroom is poisonous or nonpoisonous based on its attributes(ie, cap shape or color). Foraging for mushrooms can be a sustainable way to find food, however, with many species of mushrooms being poisonous, this can be quite risky. Thus it is important to be able to tell the poisonous from nonpoisonous mushrooms. The data set used contains 23 species of gilled mushrooms in the Agaricus and Lepiota mushroom family.

Methodology -

Data Preparation.

The original dataset contained columns of mushroom attributes, however entries were in the form of one-letter keys which correspond to values explained in the dataset. For example, ‘n’ = brown, ‘y’ = yellow, etc. We decided to one-hot-encode the data so that the information could be used with machine learning techniques. As a result, each attribute was separated into multiple columns, each titled “{attribute}_{key}”, representing the different instances of each attribute. Values were then set to 1’s and 0’s based on whether the attribute was present. The “class” column was similarly encoded and renamed to “poison” to avoid confusion.

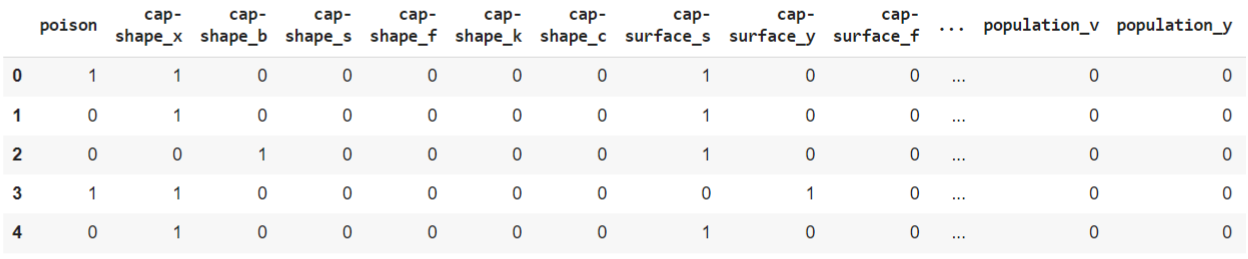
Before:



	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p

5 rows × 23 columns

After:



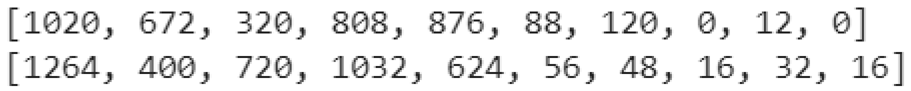
	poison	cap-shape_x	cap-shape_b	cap-shape_s	cap-shape_f	cap-shape_k	cap-shape_c	cap-surface_s	cap-surface_y	cap-surface_f	...	population_v	population_y
0	1	1	0	0	0	0	0	1	0	0	...	0	0
1	0	1	0	0	0	0	0	1	0	0	...	0	0
2	0	0	1	0	0	0	0	1	0	0	...	0	0
3	1	1	0	0	0	0	0	0	1	0	...	0	0
4	0	1	0	0	0	0	0	1	0	0	...	0	0

5 rows × 118 columns

Statistics.

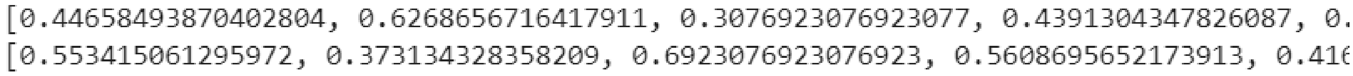
We utilized a chi-squared test to see if there was a statistically significant relationship between a mushroom's "cap-color" and whether it was poisonous or not. We figured that a mushroom's cap-color was an easy-to-spot attribute and if we were to find a relationship between the two, we could potentially use cap-color as a warning signal as to whether a mushroom is poisonous.

In this case, our null hypothesis is that there is no relationship between the "cap-color" and "poison" attributes. To perform the chi-squared test, we created a contingency table. The first row contained counts of poisonous mushrooms and the second row contained counts of nonpoisonous/edible mushrooms. The columns were separated by cap-color.



```
[1020, 672, 320, 808, 876, 88, 120, 0, 12, 0]
[1264, 400, 720, 1032, 624, 56, 48, 16, 32, 16]
```

To normalize the results, we converted the counts to percentages.



```
[0.446584938704028O4, 0.6268656716417911, 0.3076923076923077, 0.4391304347826087, 0.
[0.553415061295972, 0.373134328358209, 0.6923076923076923, 0.5608695652173913, 0.41
```

We then used the scikit-learn chi-squared test with this contingency table to get the p-value.

Machine learning.

Decision Tree: Our decision tree predicts whether a mushroom is poisonous or not based on its attributes. This was a good choice for the problem because decision trees can handle categorical data well, they are easily interpreted, and they provide intuitive visuals. We varied the max_depth parameter of the tree to prevent overfitting which was appearing in the accuracy.

Random Forests: Our random forest also predicts whether a mushroom is poisonous or not based on its attributes. We chose to do a random forest because it carries the benefits of decision trees, while also tending to have less overfitting and better generalizations. To further prevent overfitting, we varied the max_depth and max_features parameters of the forest.

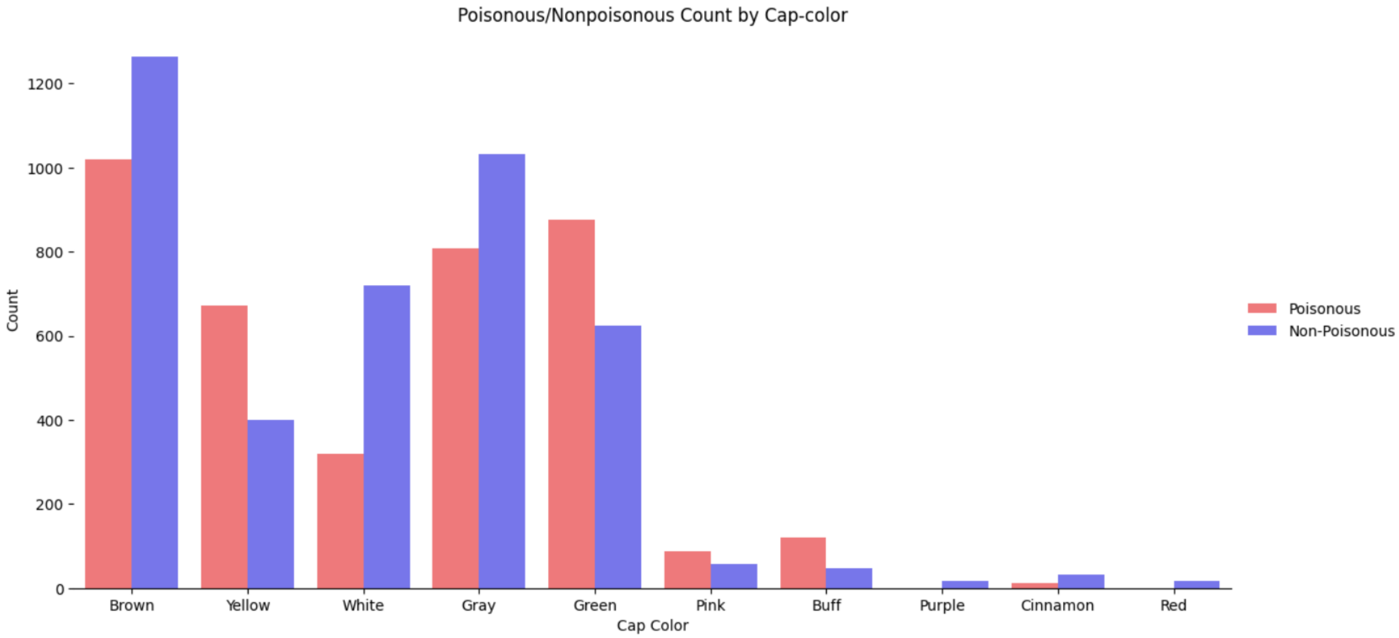
Results -

Statistics.

Chi-squared p-value: 0.983

The chi-squared test between cap-color and poison returned a p-value of 0.983, which is far too high to be considered a statistically significant relationship. We were looking for a p-value of less than 0.05. Because of this, we accept the null hypothesis and can conclude that there is no relationship between cap-color and whether a mushroom is poisonous or not.

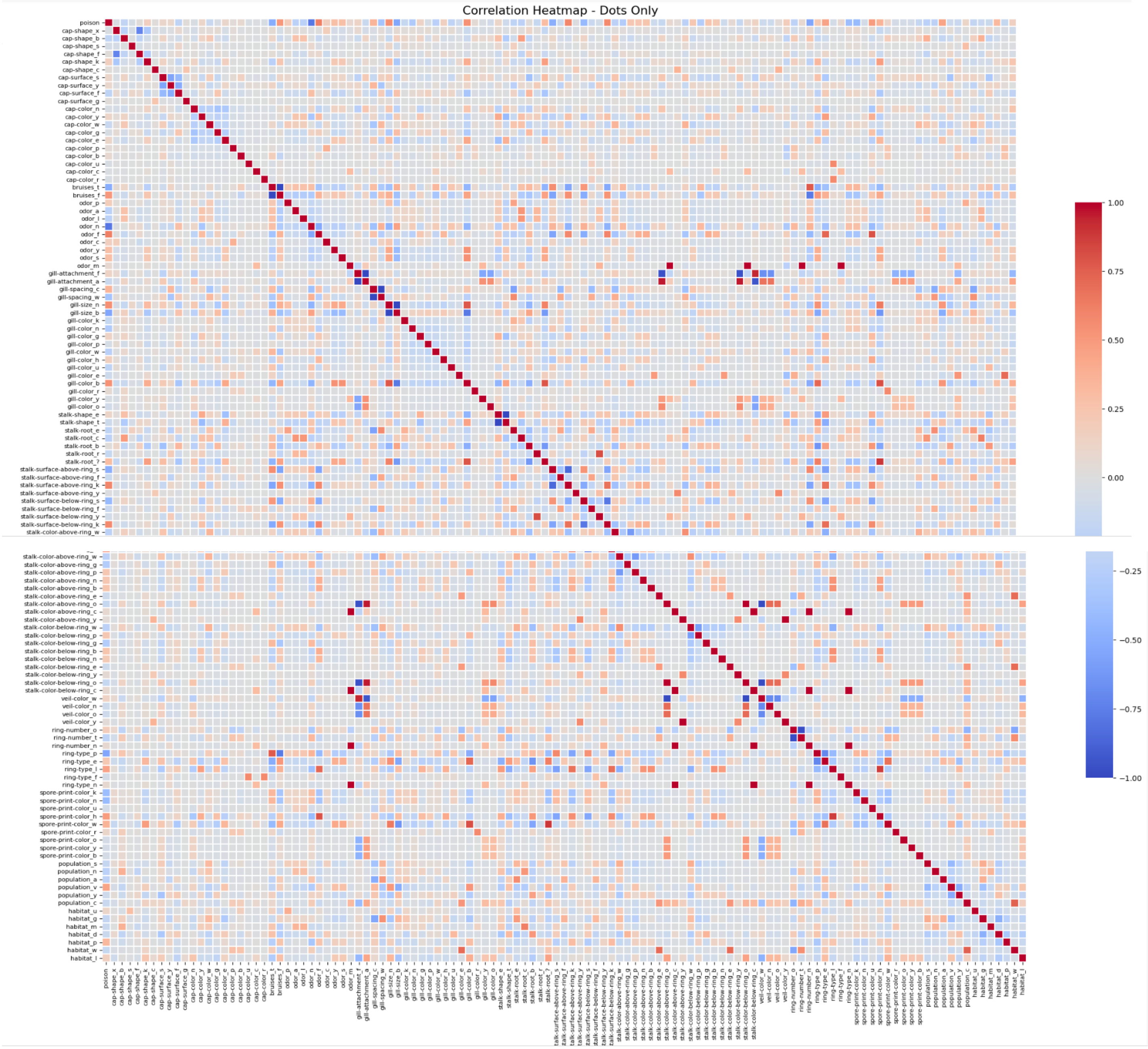
We visualized the counts of poisonous and nonpoisonous mushrooms per cap-color using a Seaborn grouped barplot:



From the grouped barplot, most cap-colors have similar counts of poisonous and nonpoisonous mushrooms. This agrees with the chi-squared test results which point towards no relationship between cap-color and poison.

We also used a Seaborn correlation heatmap against all attributes: (see next page)

Seaborn Correlation Heatmap:



As seen in the diagram, very few attributes have strong correlations with each other. In addition, the squares which represent the correlation between cap-color and poison are all gray or lighter shades. This indicates a low correlation which is similar to the results from the chi-squared test. The only strong correlation with poison appears to be with odor_n, which agrees with the most important features found in our decision forest.

Machine learning.

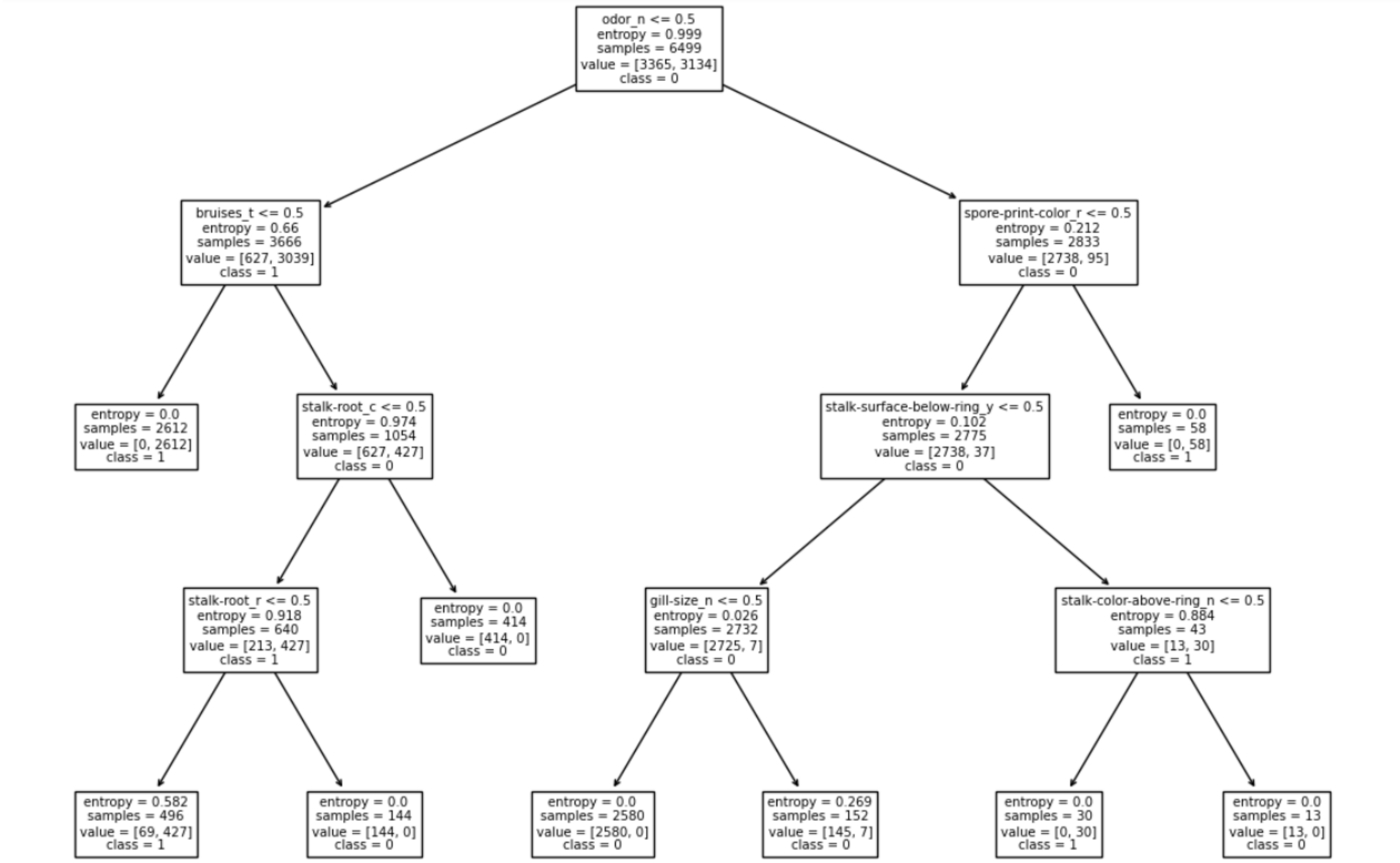
Decision Tree:

Initial Test Accuracy: 1.0

Revised Test Accuracy: 0.983

Our tree had an initial accuracy of 1.0. However, this model was likely overfitting because both the training and testing accuracies were 1.0. We combatted this by limiting the depth of the tree and setting the max_depth parameter to 4. This reduces the complexity of the tree which better allows it to capture general patterns, rather than being specific to training data. The resulting testing accuracy was 0.983.

Tree Visual:



Random Forest:

Initial Forest Accuracy: 1.0

Revised Forest Accuracy: 0.987

Similar to the trees, our forest initially had an accuracy of 1.0 which may have been overfitting because of the 1.0 accuracy of both training and testing accuracies. We combatted this by setting the max_depth parameter to 4 and the max_features to 5. Similar to the decision tree, this limits overfitting by reducing complexity and allowing general patterns to be better captured. With the random forests, we also used a validation set to limit overfitting. The revised random forest accuracy was 0.987. The process is random, but the accuracies typically lie between the range of 0.98 to 0.99.

Using the feature importances of the forest, we found that odor_n, odor_f, stalk-surface-below-ring_k, gill-size_b, and gill-color_b tended to be the in the top importances considered in the forests. However, there is a certain amount of randomness due to the nature of the forests. Despite this, odor_n (no odor) and odor_f (foul odor) were always in the top 3 most important features during testing.

Conclusions -

Based on the high accuracies of the decision tree and random forest, we can conclude that it is possible to determine with a high degree of certainty whether gilled mushrooms from the Agaricus and Lepiota mushroom family are poisonous or not, based on their attributes. This requires consideration of all of the mushroom's attributes. If only using one attribute, it is highly unlikely to be able to predict whether a mushroom is poisonous or not, as seen in the correlation heat map. However, if only one attribute could be chosen, the best bet would be to use odor_n. An odorless mushroom would be very likely nonpoisonous, based on the statistics.

Generative AI-

We asked chat gpt to help us with our visualizations. We used the code it provided for a heat map and adjusted it to fit our model.

Creating a heatmap in Python can be done using various libraries, but one popular choice is the `matplotlib` library along with its `imshow` function. Another useful library is `seaborn`, which provides a high-level interface for drawing attractive and informative statistical graphics. Here, I'll show you how to create a simple heatmap using both `matplotlib` and `seaborn`.

Using Matplotlib:

python

Copy code

```
import numpy as np
import matplotlib.pyplot as plt

# Sample data
data = np.random.random((5, 5))

# Create a heatmap
plt.imshow(data, cmap='viridis', interpolation='nearest')
plt.colorbar()

plt.show()
```

In this example, `data` is a 2D NumPy array representing the values for each cell in the heatmap. `cmap` specifies the color map, and `interpolation` determines how the colors should be interpolated between cells.

Using Seaborn:

python

Copy code

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
data = np.random.random((5, 5))

# Create a heatmap using seaborn
sns.heatmap(data, cmap='viridis', annot=True, fmt=".2f")

plt.show()
```

The `seaborn.heatmap` function is convenient as it can handle both the data and the plotting in one step. The `annot=True` option adds annotations (values) to each cell, and `fmt=".2f"` formats the annotations as floating-point numbers with two decimal places.

Feel free to adjust the code based on your specific needs and data structure. Additionally, you might need to install `matplotlib` and `seaborn` if you haven't already:

bash

Copy code

```
pip install matplotlib seaborn
```

Choose the library and style that best fits your requirements.

Also asked how to display things more nicely:

E

You

what command would i use to display the data table in a nice way: import pandas as pd

import numpy as np

df = pd.read_csv('mushrooms.csv')

df.head()

df = pd.read_csv('mushrooms.csv')

df.head()

y = df['class']

X = df.drop('class', axis=1)

```
def one_hot_columns(df_column):
    word_dict = {}
    for column in df_column.columns:
        unique_categories = df_column[column].unique()
        for category in unique_categories:
            col_name = f"{column}_{category}"
            word_present_list = df_column[column] == category
            word_dict[col_name] = word_present_list.astype(int)
    return pd.DataFrame(word_dict)

df = one_hot_columns(X)


df.insert(0, 'class', y)

df.rename(columns={'class': 'eat'}, inplace=True)

df.loc[df.eat == "p", "eat"] = 1
df.loc[df.eat == "e", "eat"] = 0
```

pd.options.display.max_columns = None

df.head

ChatGPT

ChatGPT

To display the data table in a nice way, you can use the `display` function from the IPython library if you're working in a Jupyter notebook. If you're using a script or another environment, you can simply use the `print` function. Here's an example using the `display` function:

python

Copy code

```
from IPython.display import display



# Your existing code...



pd.options.display.max_columns = None



# Display the DataFrame

display(df.head())
```

If you're not using a Jupyter notebook, you can use the `print` function:

python

Copy code

```
# Your existing code...
```

```
pd.options.display.max_columns = None



# Print the DataFrame

print(df.head())
```

Either way, this will print the first few rows of your DataFrame in a nice tabular format, making it easier to inspect the data.

We also asked chat gpt how to create a random forest and adjusted that code:

ChatGPT
To create a Random Forest in Python, you can use the `RandomForestClassifier` or `RandomForestRegressor` classes from the popular scikit-learn library. Here's a basic example using the `RandomForestClassifier` for a classification task:

python

Copy code

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

# Load the iris dataset (you can replace this with your own dataset)
iris = load_iris()
X, y = iris.data, iris.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

```
# Create a Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100,
random_state=42)

# Train the classifier
rf_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_classifier.predict(X_test)

# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

In this example:

- We use the Iris dataset, which is included in scikit-learn, but you can replace it with your own dataset.
- The data is split into training and testing sets using `train_test_split`.
- We create a `RandomForestClassifier` with 100 trees (`n_estimators=100`) and train it on the training set.
- The model is then used to make predictions on the test set, and we evaluate its accuracy.

For a regression task, you would use `RandomForestRegressor` instead.

Make sure to install scikit-learn if you haven't already:

bash

Copy code

```
pip install scikit-learn
```

Adjust the code according to your specific dataset and task. You can also tune hyperparameters like `n_estimators`, `max_depth`, and others based on your data and requirements.

The last thing we asked was how to fine tune our models:

how would on fine tune a machine learning model like a decision tree or random forest

ChatGPT

ChatGPT

Fine-tuning a machine learning model, such as a decision tree or random forest, involves optimizing its hyperparameters to improve performance. Hyperparameters are settings that are not learned from the data but are set prior to the training process. Here are steps you can follow to fine-tune a decision tree or random forest:

1. Split the Data:

Split your dataset into training and validation (or testing) sets. This allows you to train the model on one subset and evaluate its performance on another.

python

Copy code

```
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

2. Choose Hyperparameters to Tune:

Identify the hyperparameters you want to tune. For decision trees and random forests, common hyperparameters include:

- `max_depth`: Maximum depth of the tree.
- `n_estimators`: Number of trees in the forest (for random forests).
- `min_samples_split`: The minimum number of samples required to split an internal node.
- `min_samples_leaf`: The minimum number of samples required to be at a leaf node.
- `max_features`: The number of features to consider when looking for the best split.

3. Use Cross-Validation:

Perform cross-validation on the training set to evaluate different combinations of hyperparameters. This helps you identify the set of hyperparameters that generalizes well to new data.

python

Copy code

```
from sklearn.model_selection import GridSearchCV

# Example for Random Forest
param_grid = {
 'n_estimators': [50, 100, 150],
 'max_depth': [None, 10, 20],
 'min_samples_split': [2, 5, 10],
 'min_samples_leaf': [1, 2, 4],
}

rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5,
scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)
```

4. Evaluate on Validation Set:

Train the model using the best hyperparameters found during cross-validation and evaluate its performance on the validation set.

python

Copy code

```
best_rf = RandomForestClassifier(**best_params, random_state=42)
best_rf.fit(X_train, y_train)
```

```
y_val_pred = best_rf.predict(X_val)
validation_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation Accuracy: {validation_accuracy:.2f}")
```

5. Optional: Evaluate on Test Set:

After fine-tuning, you may want to evaluate the model on a separate test set that was not used during hyperparameter tuning.

6. Adjust as Needed:

Based on the performance on the validation/test set, adjust the hyperparameters and repeat the process if necessary.

Remember that fine-tuning should be done carefully, and overfitting to the validation set should be avoided. Cross-validation helps in obtaining a more robust estimate of model performance.