

Project_3_Weihua_PAN1

December 14, 2023

IE 7300: Statistical learning for Engineering

Project check point #3

Weihua Pan

206-822-8347

pan.weih@northeastern.edu

Percentage of Effort Contributed by Student : 100% Signature of Student : Weihua Pan Submission Date: 11/28/2023

```
[ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

pd.set_option('display.max_columns',30)
```

1 a) Problem Statement

1.1 Clearly define the problem statement

Problem statement : To determine whether some features of Mushroom,such as cap diameter, cap shape and cap color,can be used to predict its edibility.

1.2 State your hypothesis

Hypothesis : Mushrooms with specify feature characteristic are more likely to be edible or poisonous.

2 Dataset

2.1 Present the dataset and include a data dictionary

```
[ ]: df = pd.read_csv("MushroomDataset/secondary_data.csv",sep=";")
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61069 entries, 0 to 61068
```

Data columns (total 21 columns):

#	Column	Non-Null Count	Dtype
0	class	61069 non-null	object
1	cap-diameter	61069 non-null	float64
2	cap-shape	61069 non-null	object
3	cap-surface	46949 non-null	object
4	cap-color	61069 non-null	object
5	does-bruise-or-bleed	61069 non-null	object
6	gill-attachment	51185 non-null	object
7	gill-spacing	36006 non-null	object
8	gill-color	61069 non-null	object
9	stem-height	61069 non-null	float64
10	stem-width	61069 non-null	float64
11	stem-root	9531 non-null	object
12	stem-surface	22945 non-null	object
13	stem-color	61069 non-null	object
14	veil-type	3177 non-null	object
15	veil-color	7413 non-null	object
16	has-ring	61069 non-null	object
17	ring-type	58598 non-null	object
18	spore-print-color	6354 non-null	object
19	habitat	61069 non-null	object
20	season	61069 non-null	object

dtypes: float64(3), object(18)

memory usage: 9.8+ MB

```
[ ]:  class  cap-diameter  cap-shape  cap-surface  cap-color  does-bruise-or-bleed  \
0      p           15.26          x           g           o           f
1      p           16.60          x           g           o           f
2      p           14.07          x           g           o           f
3      p           14.17          f           h           e           f
4      p           14.64          x           h           o           f

      gill-attachment  gill-spacing  gill-color  stem-height  stem-width  stem-root  \
0                   e           NaN          w           16.95       17.09          s
1                   e           NaN          w           17.99       18.19          s
2                   e           NaN          w           17.80       17.74          s
3                   e           NaN          w           15.77       15.98          s
4                   e           NaN          w           16.53       17.20          s

      stem-surface  stem-color  veil-type  veil-color  has-ring  ring-type  \
0                y           w           u           w           t           g
1                y           w           u           w           t           g
2                y           w           u           w           t           g
3                y           w           u           w           t           p
4                y           w           u           w           t           p
```

	spore-print-color	habitat	season
0	NaN	d	w
1	NaN	d	u
2	NaN	d	w
3	NaN	d	w
4	NaN	d	w

The dataset contains 61069 rows and 21 columns. The target variable is class (e or p).

```
[ ]: df.select_dtypes(include='object')
```

```
[ ]:
class cap-shape cap-surface cap-color does-bruise-or-bleed \
0          p          x          g          o          f
1          p          x          g          o          f
2          p          x          g          o          f
3          p          f          h          e          f
4          p          x          h          o          f
...
61064      p          s          s          y          f
61065      p          f          s          y          f
61066      p          s          s          y          f
61067      p          f          s          y          f
61068      p          s          s          y          f

gill-attachment gill-spacing gill-color stem-root stem-surface \
0              e          NaN          w          s          y
1              e          NaN          w          s          y
2              e          NaN          w          s          y
3              e          NaN          w          s          y
4              e          NaN          w          s          y
...
61064          f          f          f          NaN          NaN
61065          f          f          f          NaN          NaN
61066          f          f          f          NaN          NaN
61067          f          f          f          NaN          NaN
61068          f          f          f          NaN          NaN

stem-color veil-type veil-color has-ring ring-type spore-print-color \
0          w          u          w          t          g          NaN
1          w          u          w          t          g          NaN
2          w          u          w          t          g          NaN
3          w          u          w          t          p          NaN
4          w          u          w          t          p          NaN
...
61064          y          NaN          NaN          f          f          NaN
61065          y          NaN          NaN          f          f          NaN
```

61066	y	NaN	NaN	f	f	NaN
61067	y	NaN	NaN	f	f	NaN
61068	y	NaN	NaN	f	f	NaN

	habitat	season
0	d	w
1	d	u
2	d	w
3	d	w
4	d	w
...
61064	d	a
61065	d	a
61066	d	u
61067	d	u
61068	d	u

[61069 rows x 18 columns]

17 of them are categorical feature :['cap-shape', 'cap-surface', 'cap-color', 'does-bruise-or-bleed', 'gill-attachment', 'gill-spacing', 'gill-color', 'stem-root', 'stem-surface', 'stem-color', 'veil-type', 'veil-color', 'has-ring', 'ring-type', 'spore-print-color', 'habitat', 'season'] I will do one-hot-encode later.

```
[ ]: df.select_dtypes(include='number')
```

```
[ ]:
      cap-diameter  stem-height  stem-width
0             15.26         16.95        17.09
1             16.60         17.99        18.19
2             14.07         17.80        17.74
3             14.17         15.77        15.98
4             14.64         16.53        17.20
...
61064           1.18           3.93           6.22
61065           1.27           3.18           5.43
61066           1.27           3.86           6.37
61067           1.24           3.56           5.44
61068           1.17           3.25           5.45
```

[61069 rows x 3 columns]

3 of them are numerical: ['cap-diameter', 'stem-height', 'stem-width'].

2.1.1 Dataset Dictionary

Target variable: **class** divided in edible=e and poisonous=p n: nominal, m: metrical 1. cap-diameter (m): float number in cm 2. cap-shape (n): * bell=b, * conical=c, * convex=x, * flat=f, * sunken=s, * spherical=p, * others=o 3. cap-surface (n): * fibrous=i, * grooves=g, * scaly=y, * smooth=s, * shiny=h, * leathery=l, * silky=k, * sticky=t, * wrinkled=w, * fleshy=e 4. cap-color

(n): * brown=n, * buff=b, * gray=g, * green=r, * pink=p, * purple=u, * red=e, * white=w, * yellow=y, * blue=l, * orange=o, * black=k 5. does-bruise-bleed (n): * bruises-or-bleeding=t, * no=f 6. gill-attachment (n): * adnate=a, * adnexed=x, * decurrent=d, * free=e, * sinuate=s, * pores=p, * none=f, * unknown=? 7. gill-spacing (n): * close=c, * distant=d, * none=f 8. gill-color (n): * see cap-color + * none=f 9. stem-height (m): float number in cm 10. stem-width (m): float number in mm 11. stem-root (n): * bulbous=b, * swollen=s, * club=c, * cup=u, * equal=e, * rhizomorphs=z, * rooted=r 12. stem-surface (n): see cap-surface + none=f 13. stem-color (n): see cap-color + none=f 14. veil-type (n): partial=p, universal=u 15. veil-color (n): see cap-color + none=f 16. has-ring (n): * ring=t, * none=f 17. ring-type (n): * cobwebby=c, * evanescent=e, * flaring=r, * grooved=g, * large=l, * pendant=p, * sheathing=s, * zone=z, * scaly=y, * movable=m, * none=f, * unknown=? 18. spore-print-color (n): see cap color 19. habitat (n): * grasses=g, * leaves=l, * meadows=m, * paths=p, * heaths=h, * urban=u, * waste=w, * woods=d 20. season (n): * spring=s, * summer=u, * autumn=a, * winter=w

Most of the columns are categorical, so I need to do one-hot-encoding for these categorical data in order to train a model

2.2 Explain how this dataset supports your hypothesis

```
[ ]: # split mushrooms df into poison and edible df.
poison = df[df['class']=='p']
edible = df[df['class']=='e']
```

```
[ ]: print(poison.shape)
print(edible.shape)
```

```
(33888, 21)
```

```
(27181, 21)
```

```
[ ]: p_color = poison['cap-color'].value_counts().index
p_counts = poison['cap-color'].value_counts().values
e_color = edible['cap-color'].value_counts().index
e_counts = edible['cap-color'].value_counts().values
```

```
[ ]: color_code = {"n":"brown",
                  "b":"#f0dc82", #buff
                  "g":"gray",
                  "r":"green",
                  "p":"pink",
                  "u":"purple",
                  "e":"red",
                  "w":"white",
                  "y":"yellow",
                  "l":"blue",
                  "o":"orange",
                  "k":"black"}
```

```
[ ]: color_map = {"n":"brown",
                  "b":"buff", #buff
                  "g":"gray",
                  "r":"green",
                  "p":"pink",
                  "u":"purple",
                  "e":"red",
                  "w":"white",
                  "y":"yellow",
                  "l":"blue",
                  "o":"orange",
                  "k":"black"}
```

```
[ ]: import warnings
warnings.filterwarnings('ignore')

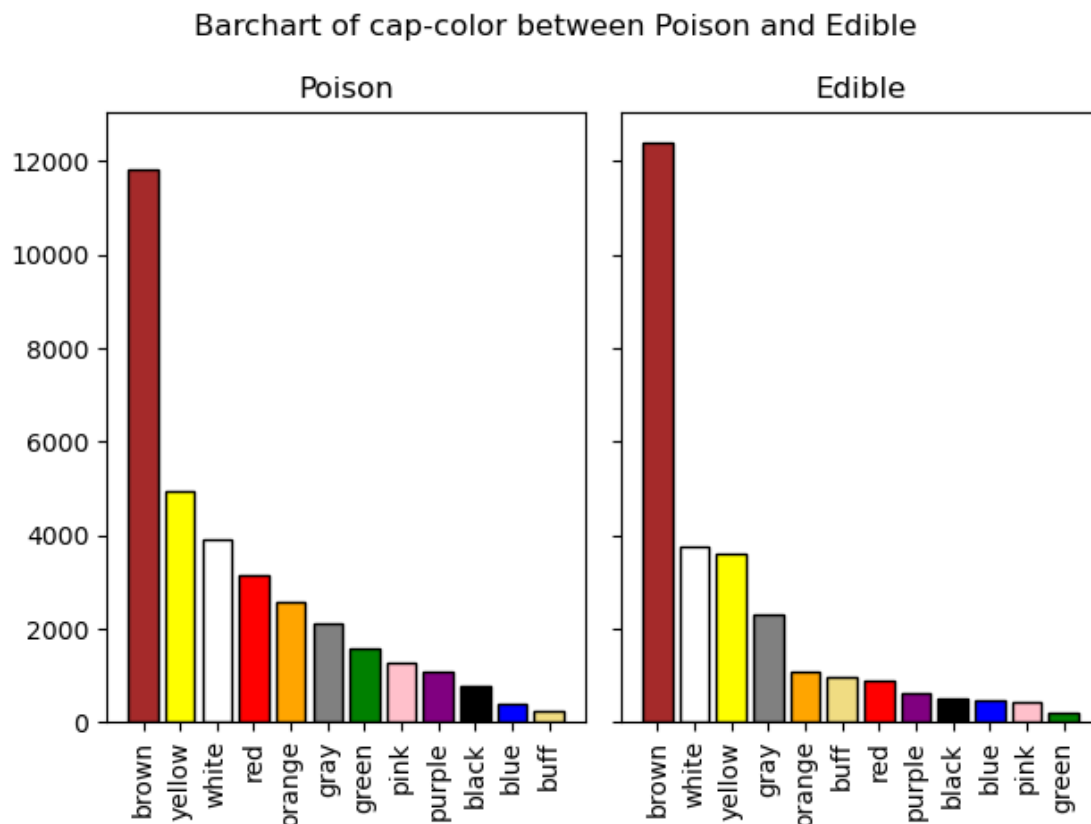
fig,ax = plt.subplots(1,2,sharey=True)

ax[0].bar([color_map[c] for c in p_color],p_counts,color=[color_code[code] for
↳code in p_color],
          edgecolor='black')
ax[0].set_title('Poison')

#
ax[1].bar([color_map[c] for c in e_color],e_counts,color=[color_code[code] for
↳code in e_color],
          edgecolor='black')
ax[1].set_title('Edible')

# rotate xlabels
ax[0].set_xticklabels([color_map[c] for c in p_color],rotation=90)
ax[1].set_xticklabels([color_map[c] for c in e_color],rotation=90)

fig.suptitle("Barchart of cap-color between Poison and Edible")
plt.tight_layout()
plt.show()
```



Using cap-color and other features, we are able to find some pattern to classify base on these features. For example, From the above plot, We can see most green and red mushroom are poisonous. In addition, the most common color is brown for both poison and edible mushrooms, Therefore, it might be tough to use brown to classify whether a mushroom is poisonous or edible.

3 Exploratory Data Analysis(EDA)

3.1 Descriptive Statistics

```
[ ]: df.describe()
```

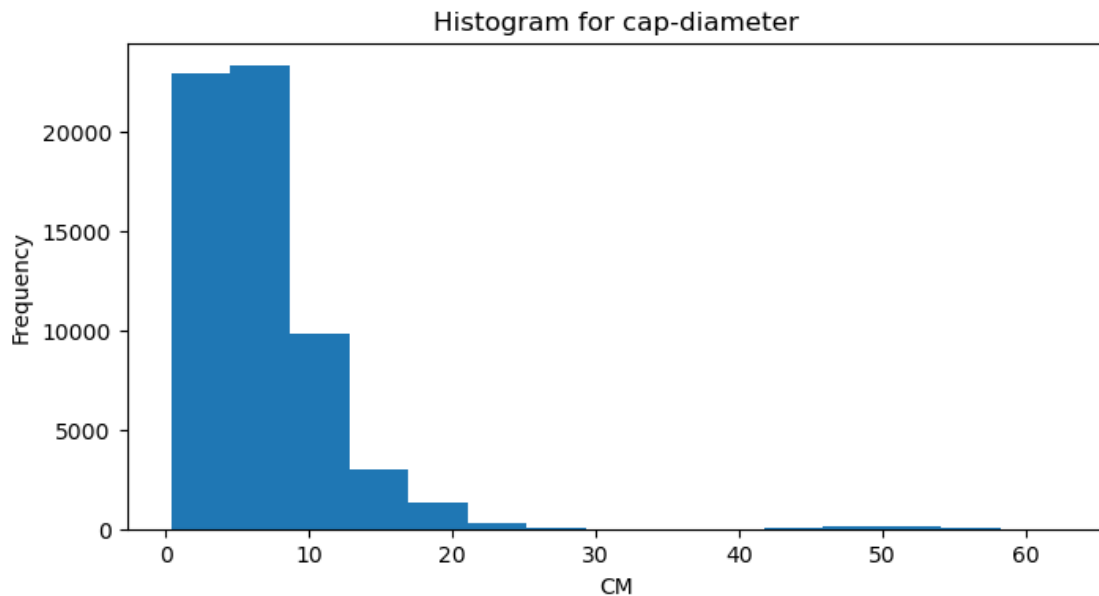
```
[ ]:
      cap-diameter  stem-height  stem-width
count  61069.000000  61069.000000  61069.000000
mean      6.733854    6.581538    12.149410
std       5.264845    3.370017    10.035955
min       0.380000    0.000000    0.000000
25%       3.480000    4.640000    5.210000
50%       5.860000    5.950000    10.190000
75%       8.540000    7.740000    16.570000
max      62.340000   33.920000   103.910000
```

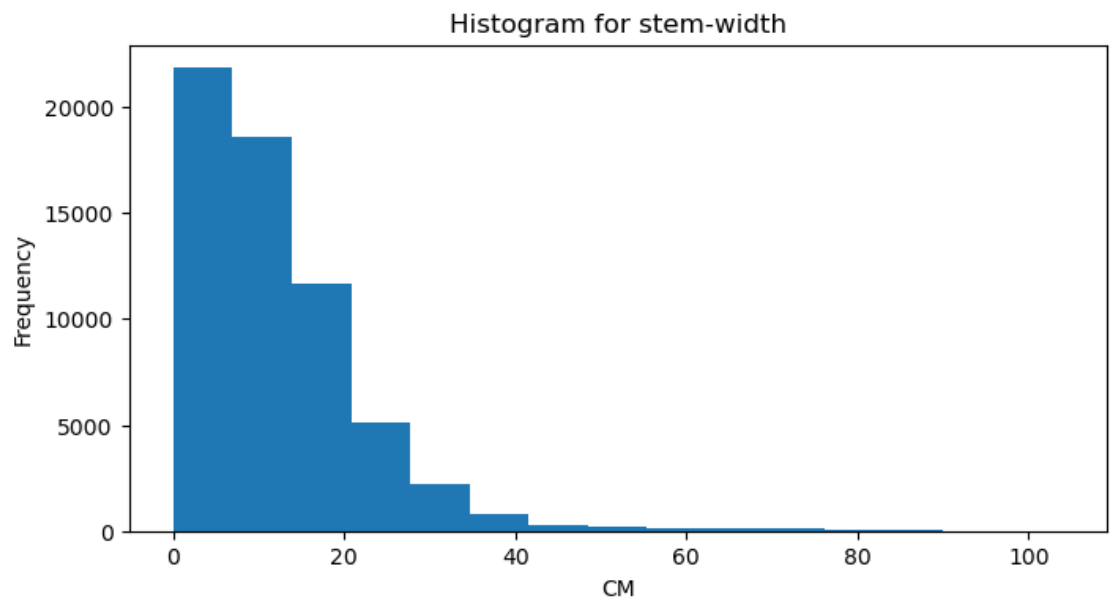
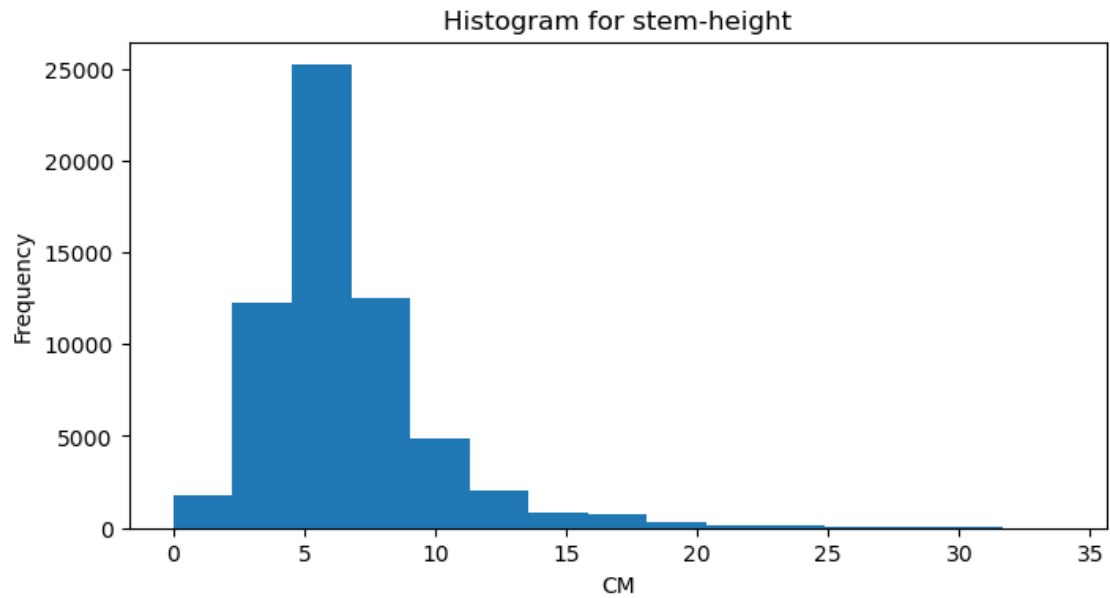
Only 3 columns are numerical. 1. cap-diameter: * min: 0.38 cm * max: 62.34 cm * mean 6.7 cm
2. stem-height: * min: 0 cm * max: 33.92 cm * mean 6.58 cm 3. stem-width: * min: 0 cm * max: 103.91 cm * mean: 10.03 cm

Base on the basic statistic, I can take a guess **cap-diameter** and **stem-height** are in right skewed, and **stem-width** is about normal by mean and 50% quartile. And they all have some outliers. I will consider whether remove them later.

```
[ ]: # draw histogram for all numerical columns
numerical_columns = df.select_dtypes(include='number')

for column in numerical_columns:
    fig, ax = plt.subplots(figsize=(8, 4))
    df[column].plot(kind='hist', title=f"Histogram for {column}", ax=ax, bins=15)
    ax.set_xlabel("CM")
    plt.show()
```





By the histogram plot, we can see `cap-diameter` and `stem-width` are right skewed. And only `stem-height` is normal distributive.

3.2 Bar chart among 2 target variables

```
[ ]: object_columns = df.select_dtypes(include=['object']).columns[1:]

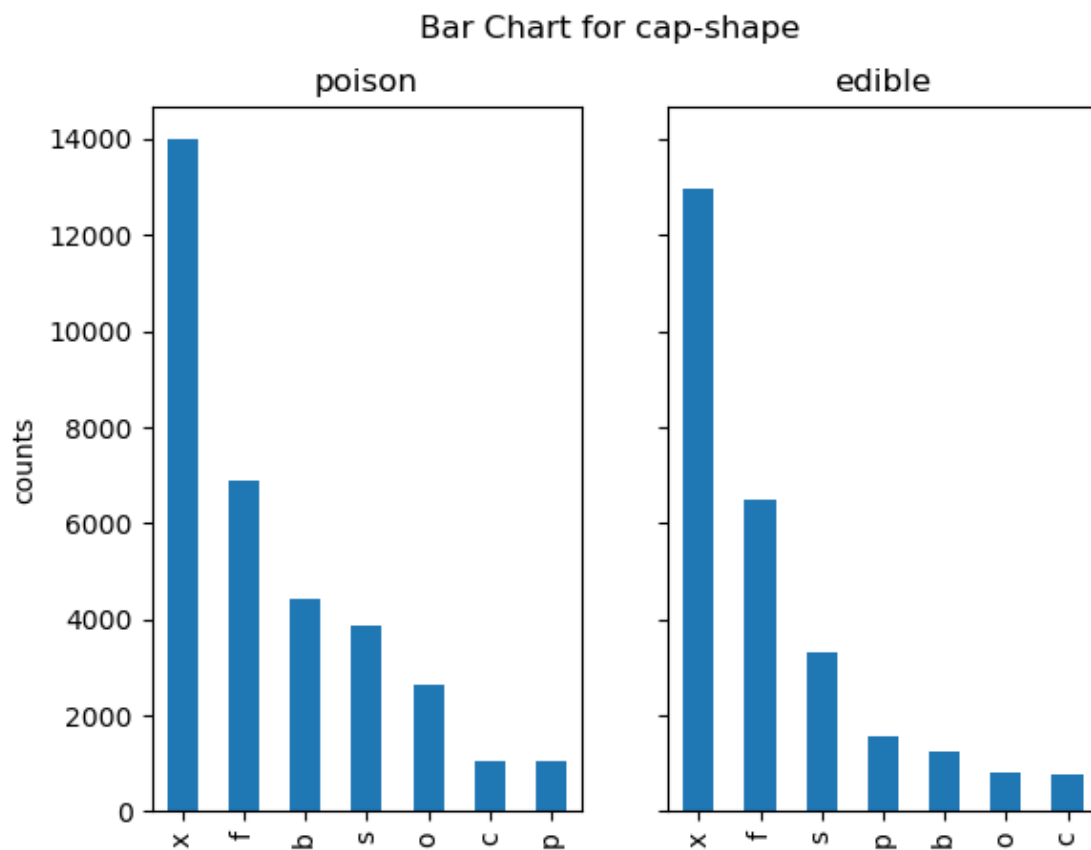
for column in object_columns:
    # Count the occurrences of each category
    poison_C = poison[column].value_counts()
    edible_C = edible[column].value_counts()

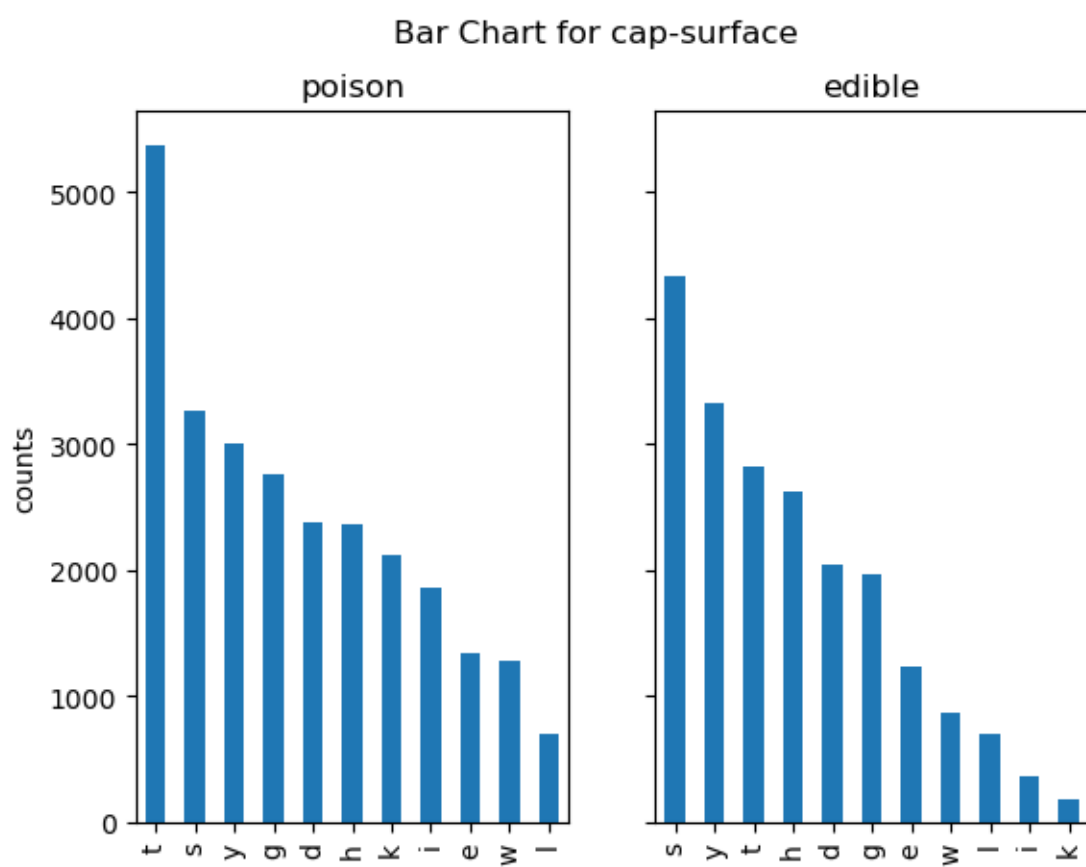
    # Create a subplot chart
    fig, ax = plt.subplots(1, 2, sharey=True)

    poison_C.plot(kind='bar', ax=ax[0], title='poison', ylabel='counts')
    edible_C.plot(kind='bar', ax=ax[1], title='edible')

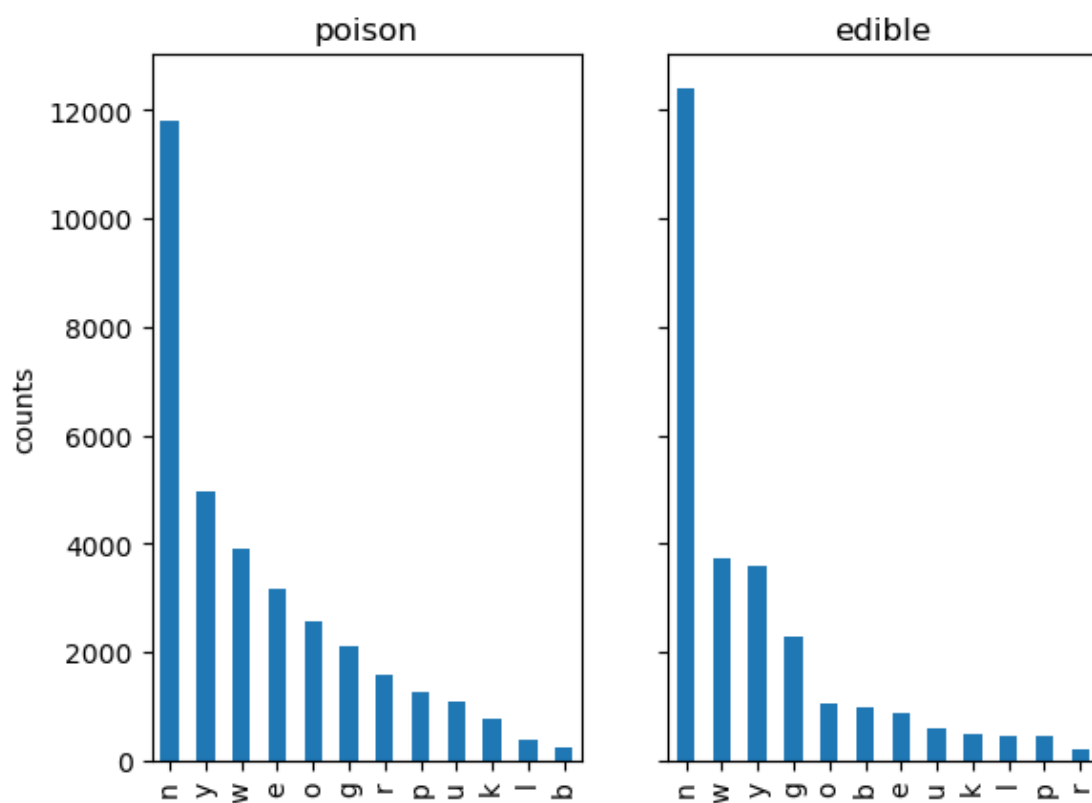
    fig.suptitle(f'Bar Chart for {column}')

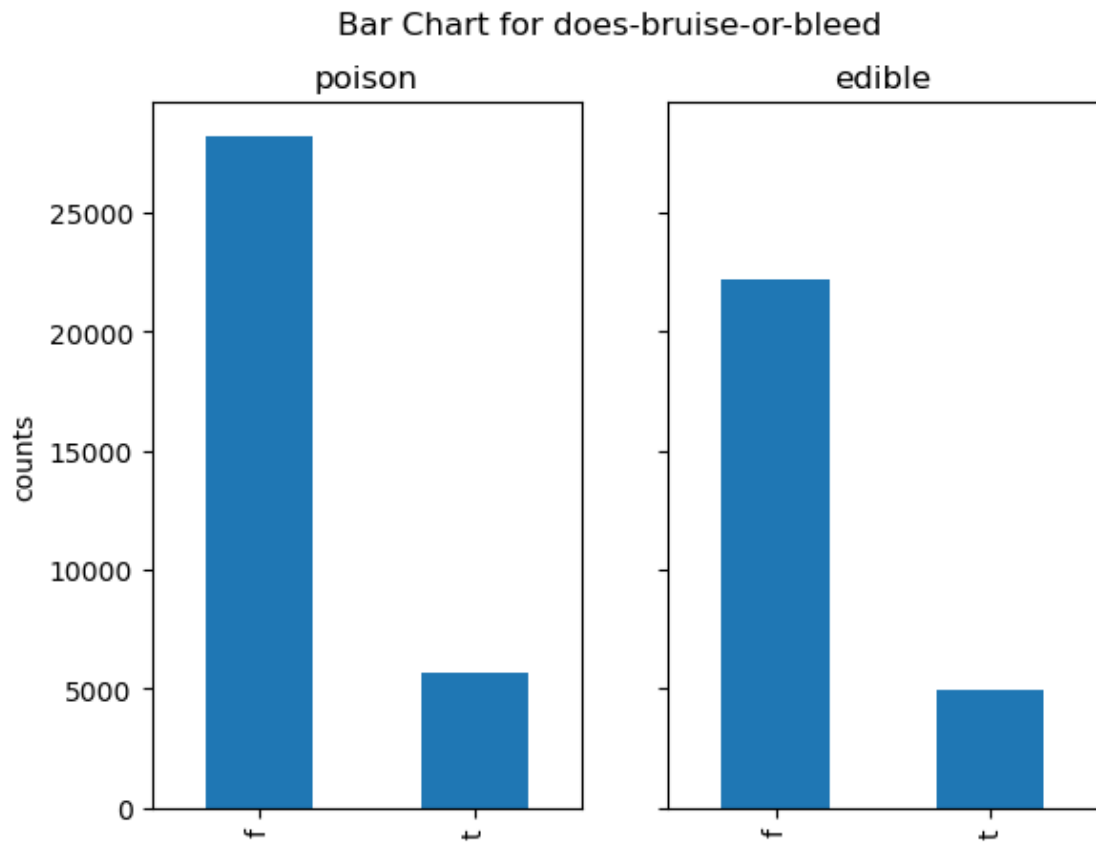
    # Show the plot
    plt.show()
```



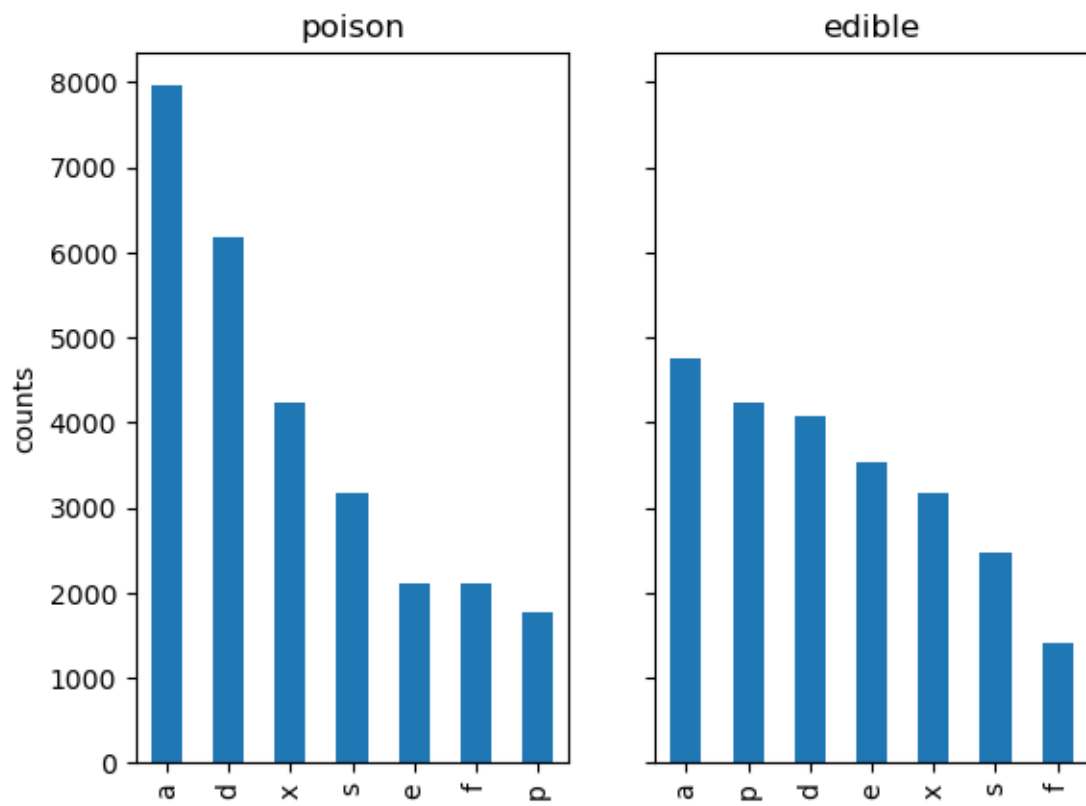


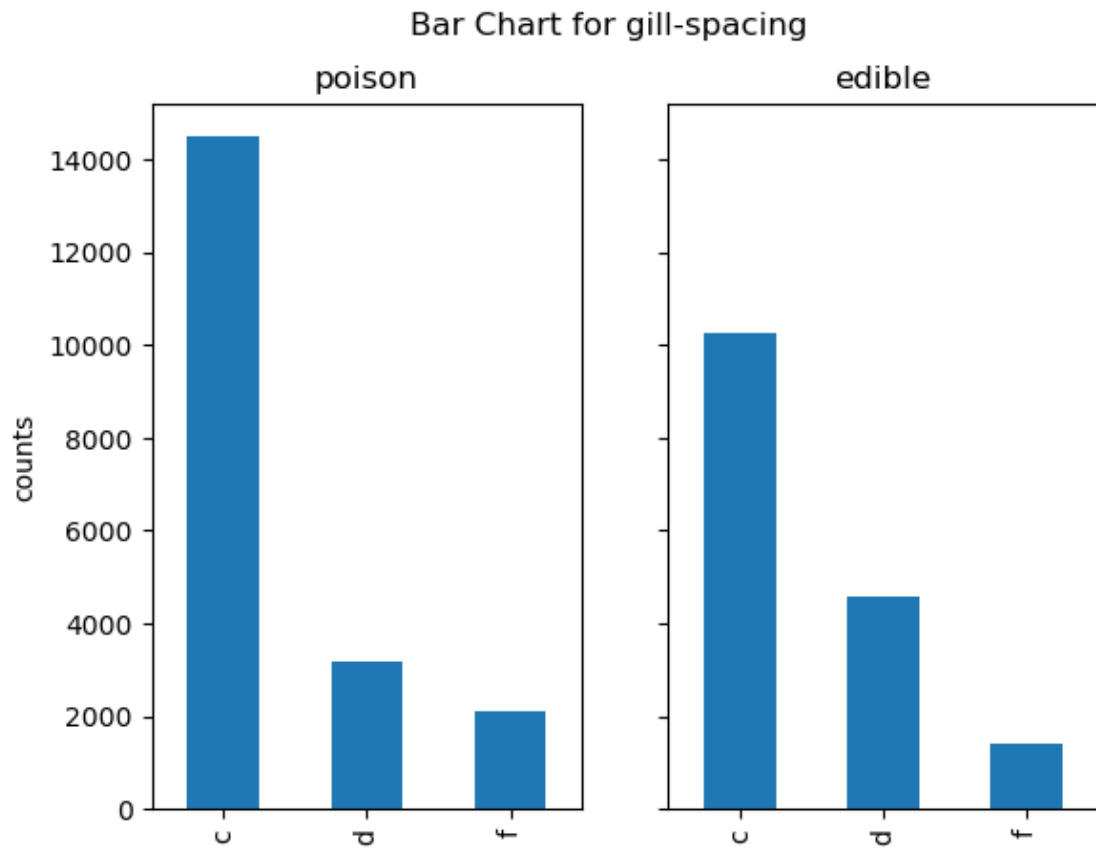
Bar Chart for cap-color



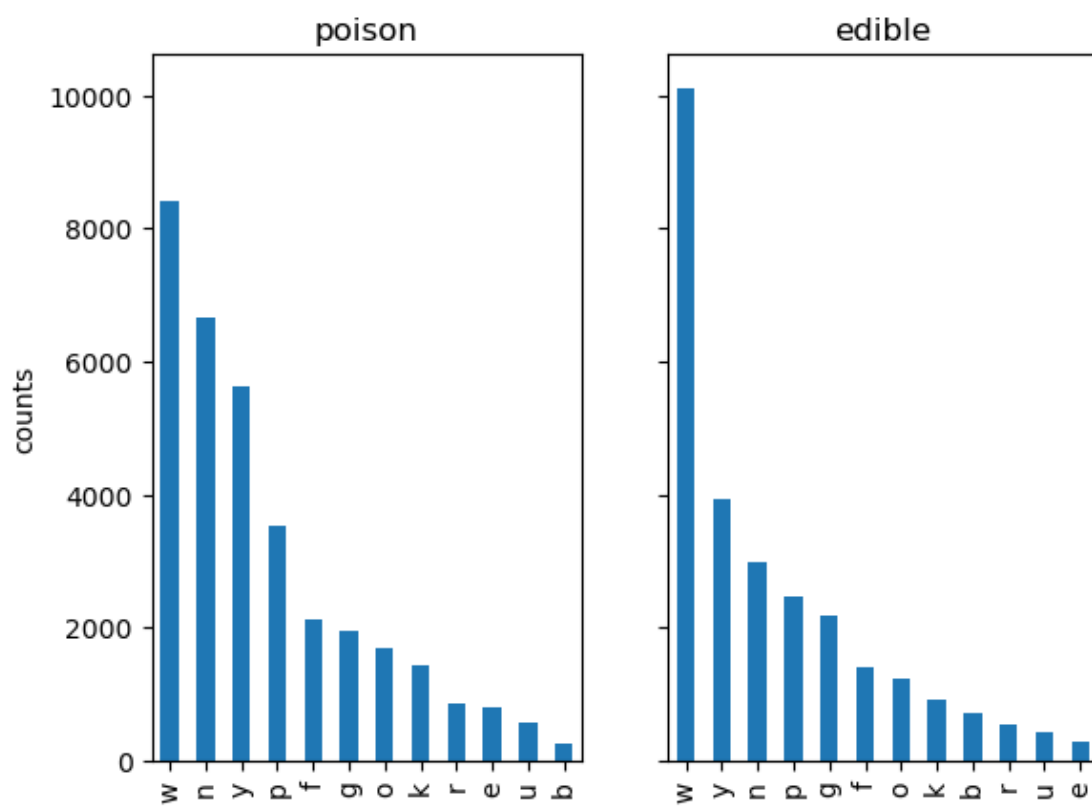


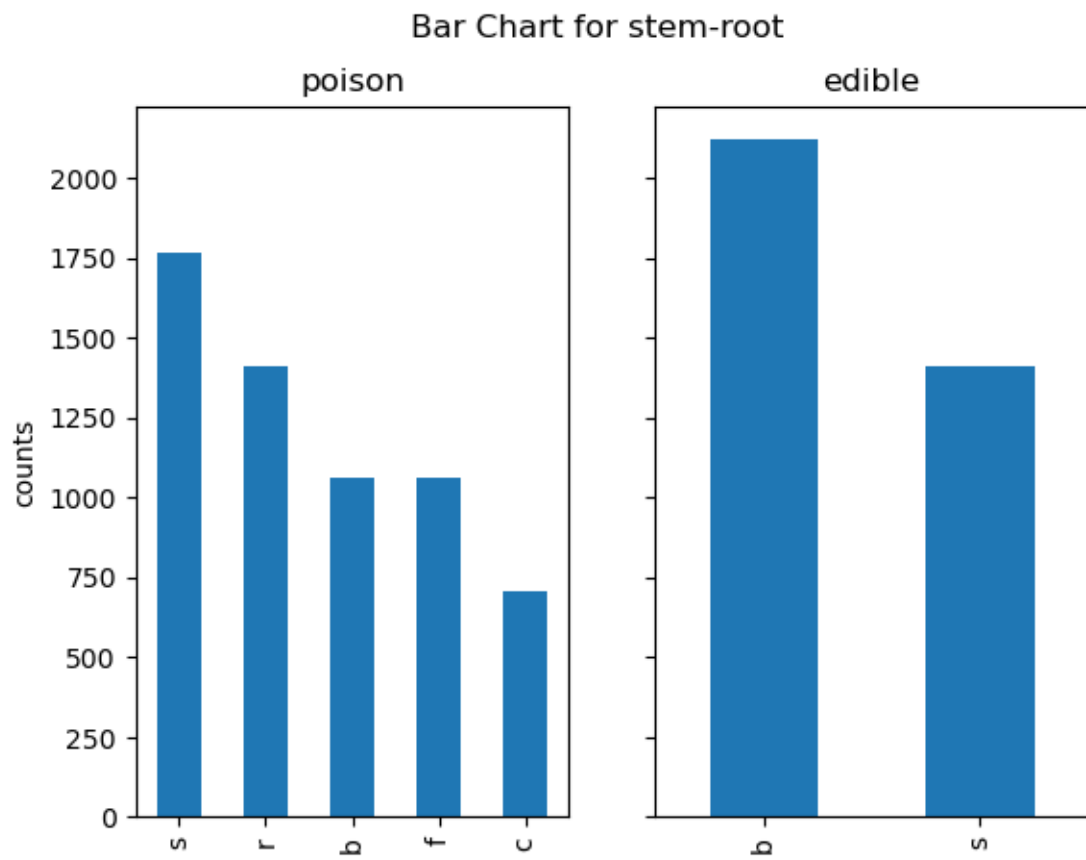
Bar Chart for gill-attachment

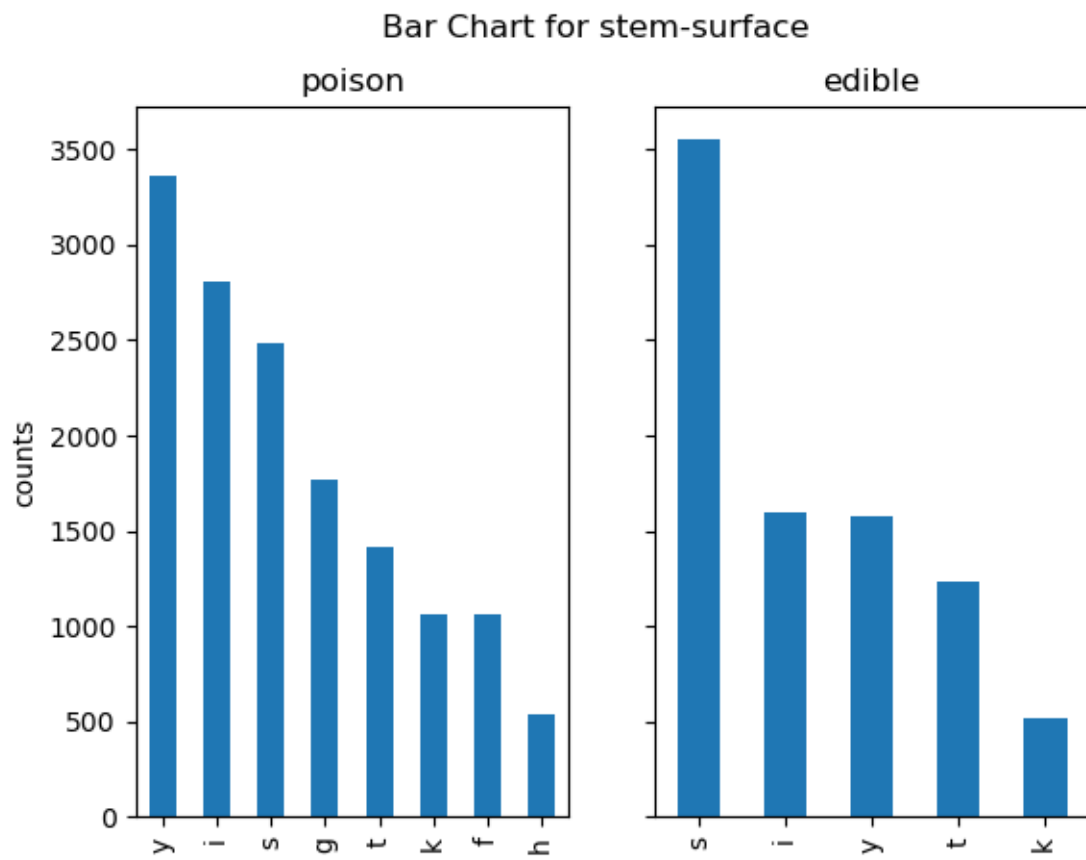




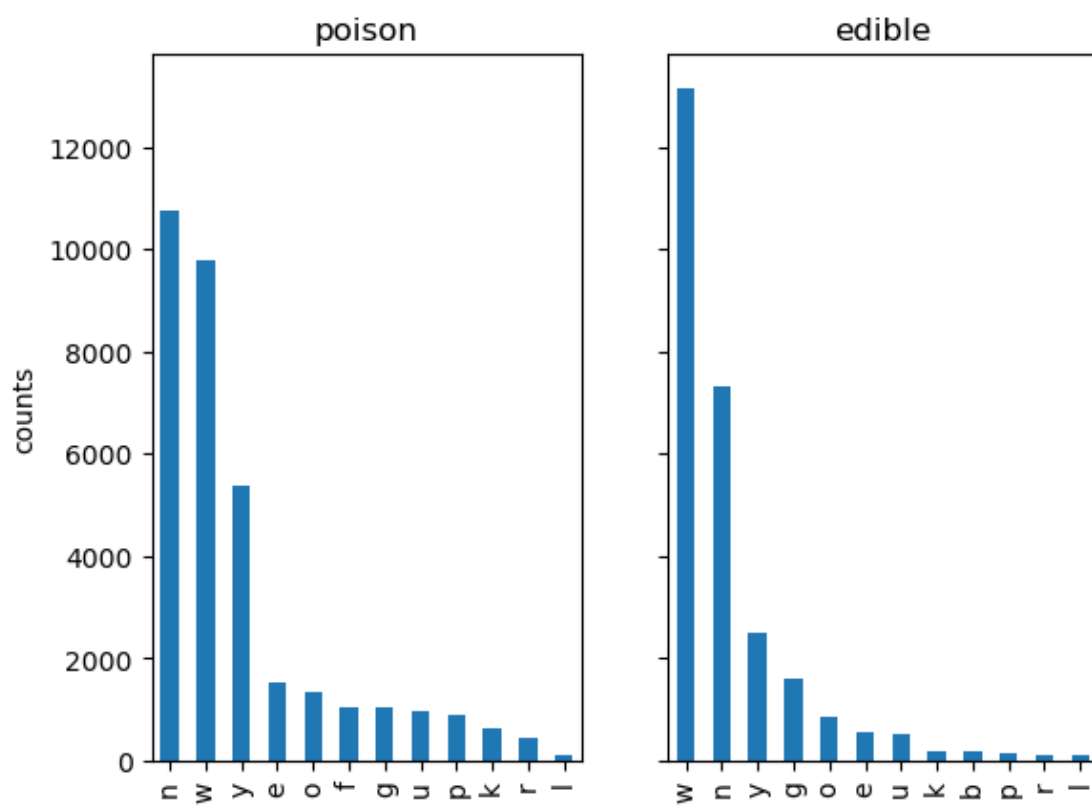
Bar Chart for gill-color

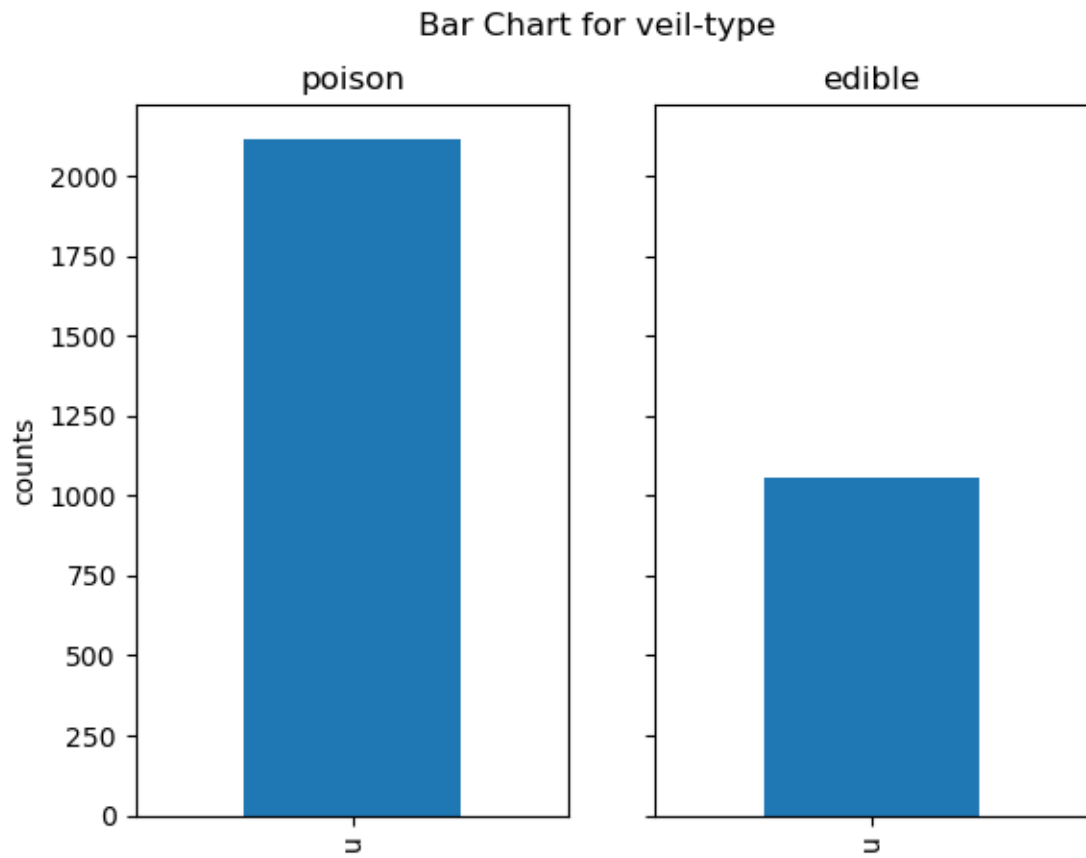


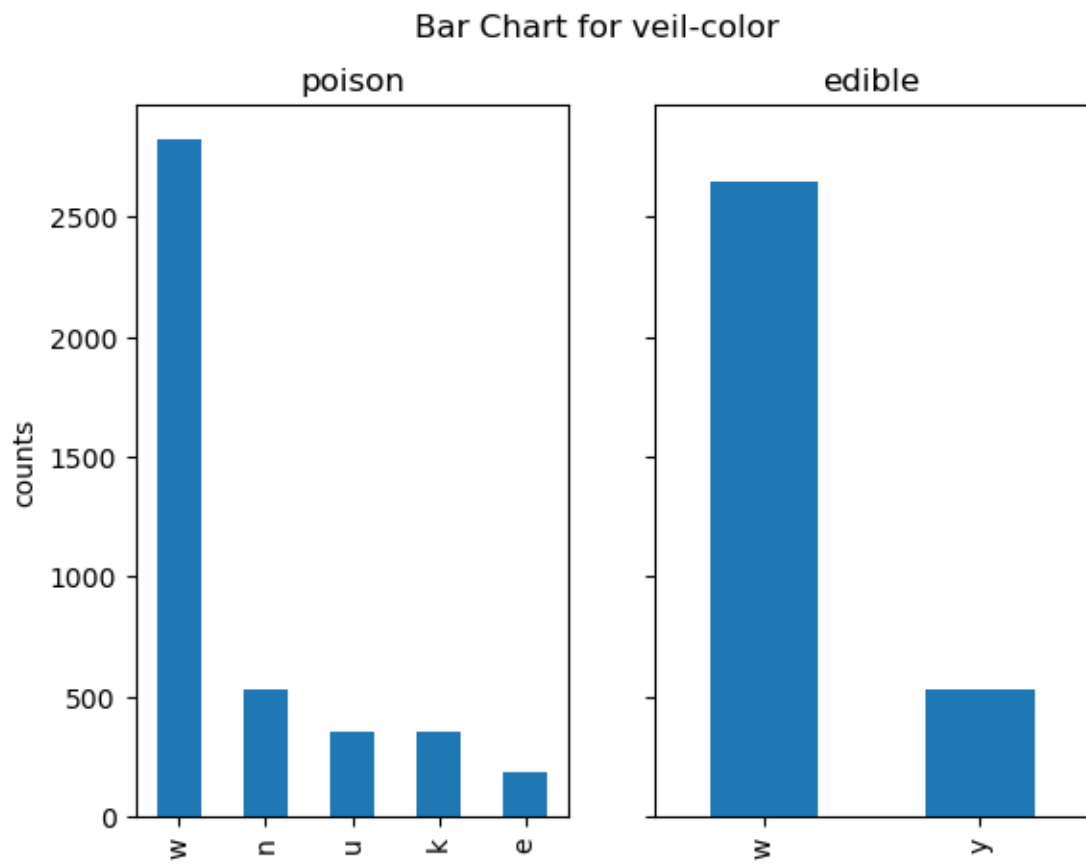


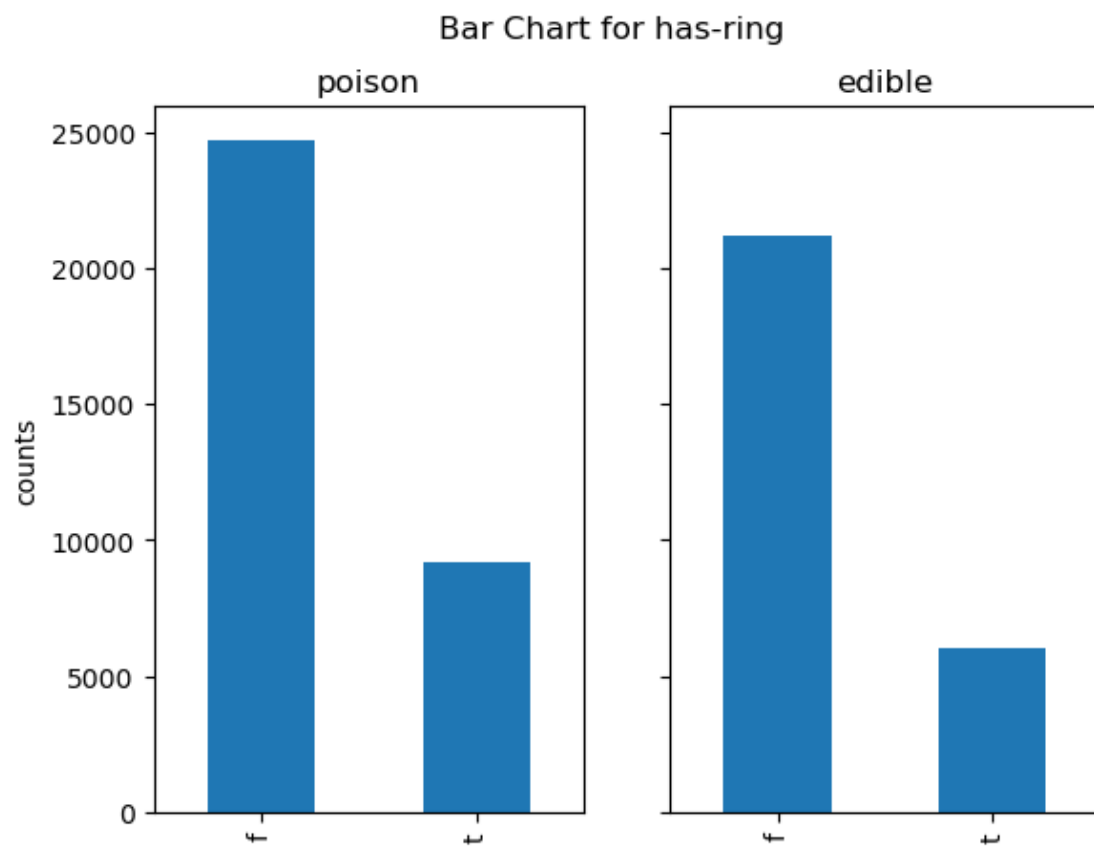


Bar Chart for stem-color

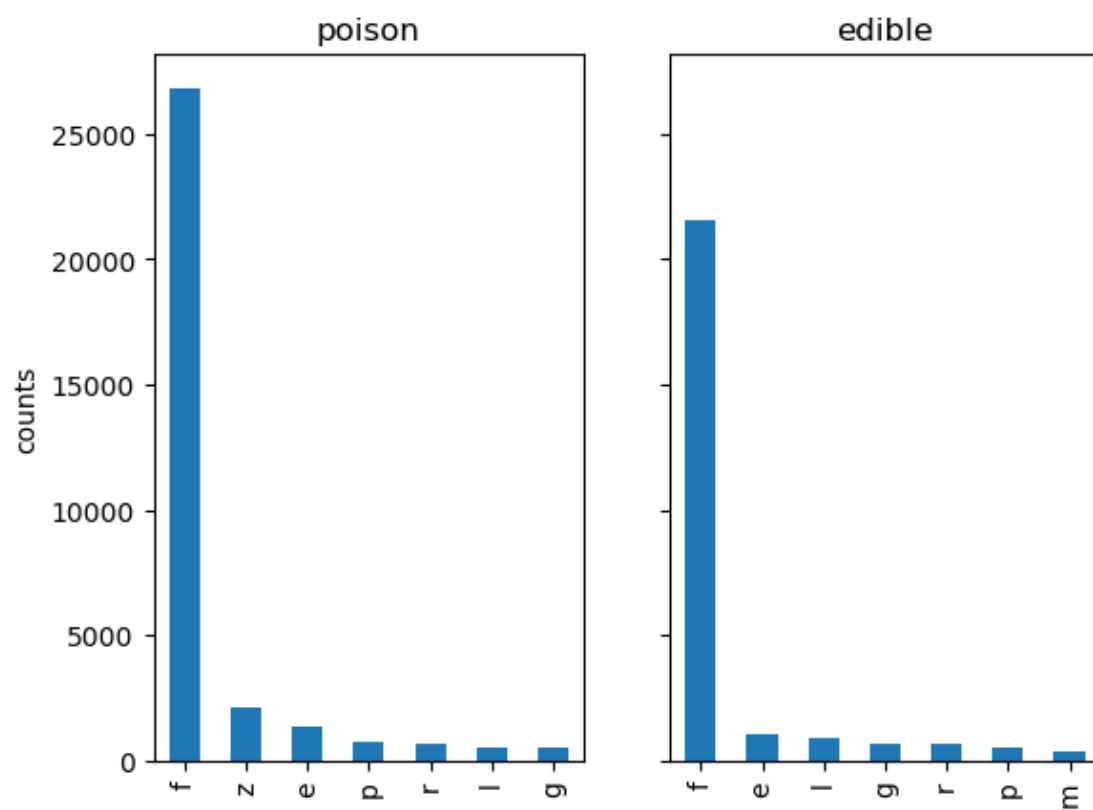


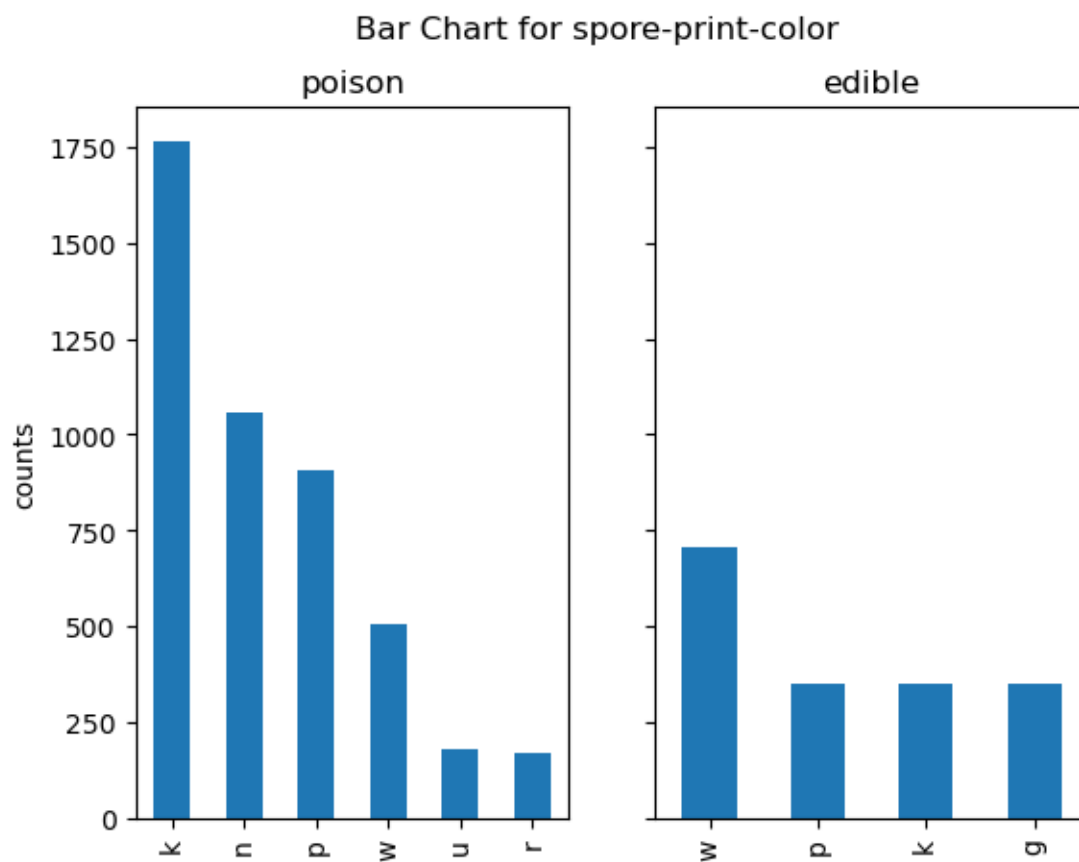




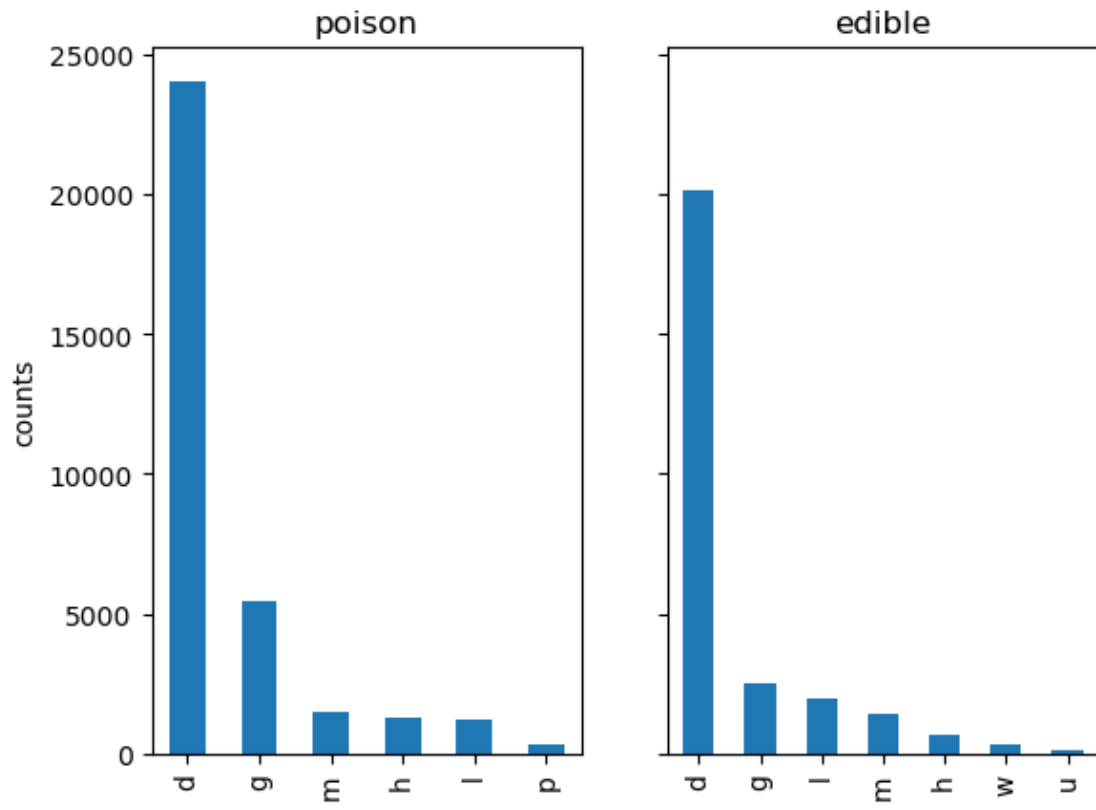


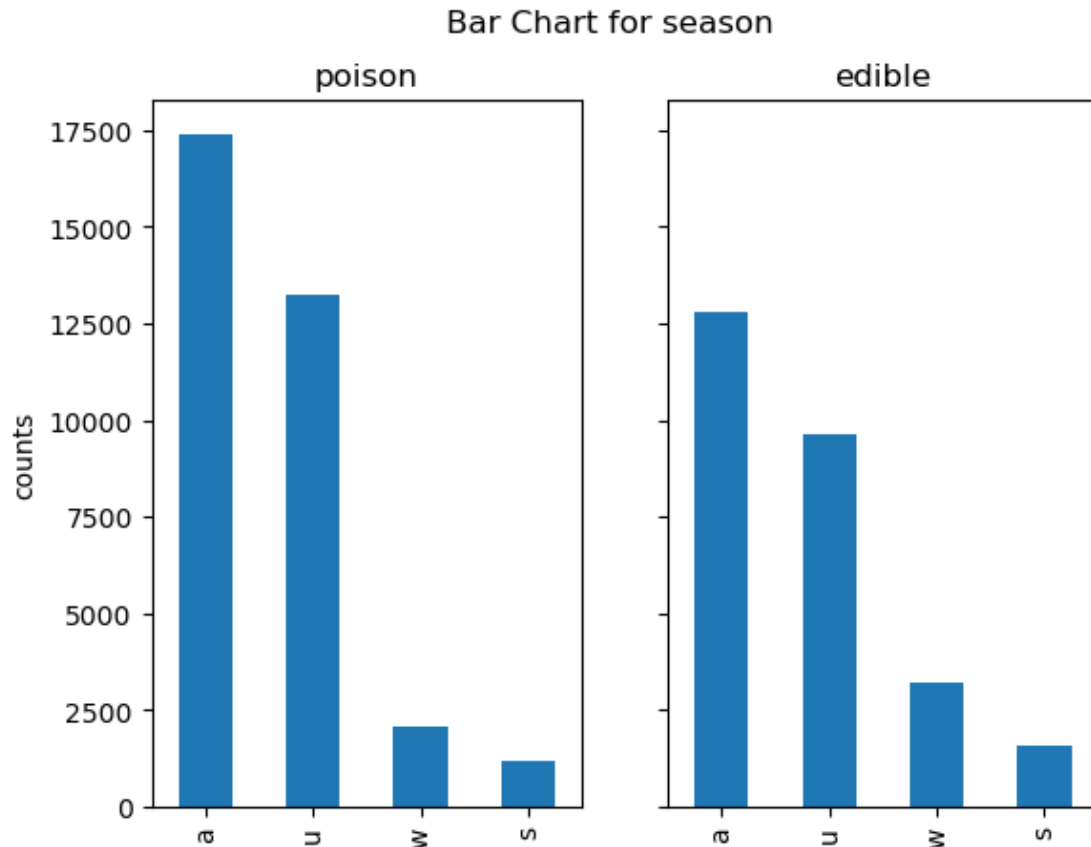
Bar Chart for ring-type





Bar Chart for habitat





By the above bar charts, we can find some pattern to classify poisonous mushroom by categorical features. For example, In **season** bar chart, we can see most poisonous mushrooms live in autumn and summer. And we can find pattern in **veil-color** as well. We can easily classify the mushrooms with yellow veil is edible since there is no poisonous mushroom with yell veil. All the **veil-type** are universal, other than that are missing values.

3.3 Missing values

```
[ ]: NA_dict = dict()
      for column in df.columns[1:]:
          NA_dict[column] = df[column].isna().sum()

      NA_dict
```

```
[ ]: {'cap-diameter': 0,
      'cap-shape': 0,
      'cap-surface': 14120,
      'cap-color': 0,
      'does-bruise-or-bleed': 0,
      'gill-attachment': 9884,
```

```

'gill-spacing': 25063,
'gill-color': 0,
'stem-height': 0,
'stem-width': 0,
'stem-root': 51538,
'stem-surface': 38124,
'stem-color': 0,
'veil-type': 57892,
'veil-color': 53656,
'has-ring': 0,
'ring-type': 2471,
'spore-print-color': 54715,
'habitat': 0,
'season': 0}

```

cap-surface,gill-attachment,gill-spacing,stem-root,stem-surface,veil-type,veil-color,ring-type,spore-print-color,habitat,season have missing values. And all of them are categorical. All the missing values here might mean unknown or not applicable. I decide to impute them using “?”, since I have no idea how these missing values appear and cannot apply imputation strategy like mode or k_nearest. We can let the model to decide and find pattern when they meet the missing values “?”.

```
[ ]: df= df.fillna('?')
```

3.4 Feature engineering

```
[ ]: y = df.iloc[:,0]
X = df.iloc[:,1:]
X = pd.get_dummies(X) # encode categorical data by one-hot-encoding
X = X.astype(float) # convert true and false to 1 and 0
X
```

```
[ ]:
```

	cap-diameter	stem-height	stem-width	cap-shape_b	cap-shape_c	\
0	15.26	16.95	17.09	0.0	0.0	
1	16.60	17.99	18.19	0.0	0.0	
2	14.07	17.80	17.74	0.0	0.0	
3	14.17	15.77	15.98	0.0	0.0	
4	14.64	16.53	17.20	0.0	0.0	
...	
61064	1.18	3.93	6.22	0.0	0.0	
61065	1.27	3.18	5.43	0.0	0.0	
61066	1.27	3.86	6.37	0.0	0.0	
61067	1.24	3.56	5.44	0.0	0.0	
61068	1.17	3.25	5.45	0.0	0.0	

	cap-shape_f	cap-shape_o	cap-shape_p	cap-shape_s	cap-shape_x	\
0	0.0	0.0	0.0	0.0	1.0	
1	0.0	0.0	0.0	0.0	1.0	

2	0.0	0.0	0.0	0.0	1.0
3	1.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	1.0
...
61064	0.0	0.0	0.0	1.0	0.0
61065	1.0	0.0	0.0	0.0	0.0
61066	0.0	0.0	0.0	1.0	0.0
61067	1.0	0.0	0.0	0.0	0.0
61068	0.0	0.0	0.0	1.0	0.0

	cap-surface_?	cap-surface_d	cap-surface_e	cap-surface_g	\
0	0.0	0.0	0.0	1.0	
1	0.0	0.0	0.0	1.0	
2	0.0	0.0	0.0	1.0	
3	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	
...	
61064	0.0	0.0	0.0	0.0	
61065	0.0	0.0	0.0	0.0	
61066	0.0	0.0	0.0	0.0	
61067	0.0	0.0	0.0	0.0	
61068	0.0	0.0	0.0	0.0	

	cap-surface_h	...	spore-print-color_r	spore-print-color_u	\
0	0.0	...	0.0	0.0	
1	0.0	...	0.0	0.0	
2	0.0	...	0.0	0.0	
3	1.0	...	0.0	0.0	
4	1.0	...	0.0	0.0	
...	
61064	0.0	...	0.0	0.0	
61065	0.0	...	0.0	0.0	
61066	0.0	...	0.0	0.0	
61067	0.0	...	0.0	0.0	
61068	0.0	...	0.0	0.0	

	spore-print-color_w	habitat_d	habitat_g	habitat_h	habitat_l	\
0	0.0	1.0	0.0	0.0	0.0	
1	0.0	1.0	0.0	0.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	
3	0.0	1.0	0.0	0.0	0.0	
4	0.0	1.0	0.0	0.0	0.0	
...	
61064	0.0	1.0	0.0	0.0	0.0	
61065	0.0	1.0	0.0	0.0	0.0	
61066	0.0	1.0	0.0	0.0	0.0	
61067	0.0	1.0	0.0	0.0	0.0	

61068		0.0	1.0	0.0	0.0	0.0
-------	--	-----	-----	-----	-----	-----

	habitat_m	habitat_p	habitat_u	habitat_w	season_a	season_s \
0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0
...
61064	0.0	0.0	0.0	0.0	1.0	0.0
61065	0.0	0.0	0.0	0.0	1.0	0.0
61066	0.0	0.0	0.0	0.0	0.0	0.0
61067	0.0	0.0	0.0	0.0	0.0	0.0
61068	0.0	0.0	0.0	0.0	0.0	0.0

	season_u	season_w
0	0.0	1.0
1	1.0	0.0
2	0.0	1.0
3	0.0	1.0
4	0.0	1.0
...
61064	0.0	0.0
61065	0.0	0.0
61066	1.0	0.0
61067	1.0	0.0
61068	1.0	0.0

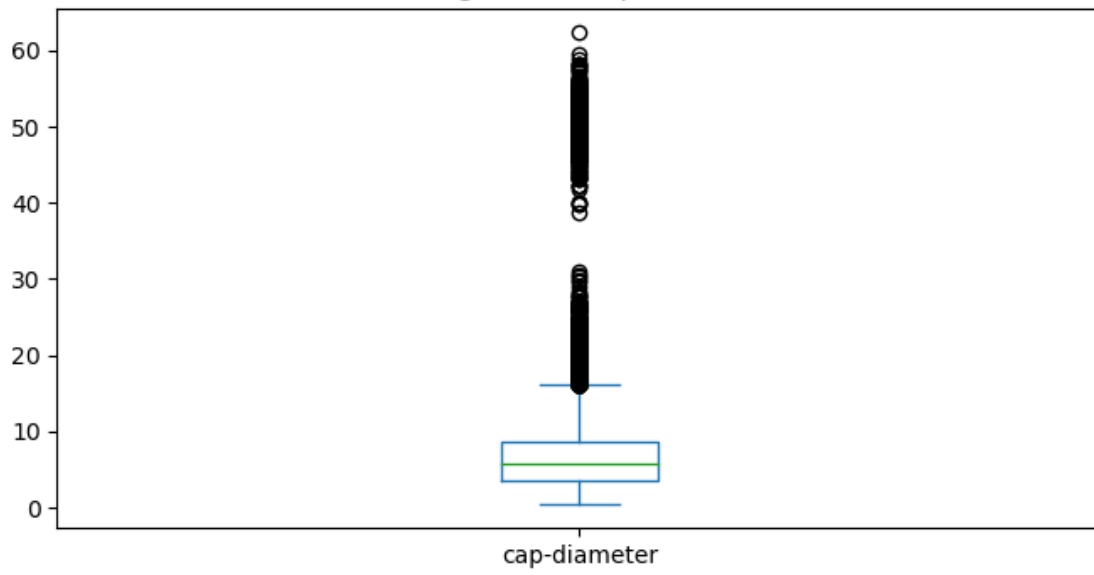
[61069 rows x 128 columns]

Most columns are categorical, so we need to one-hot-encode the categorical columns. Therefore the models can fit the dataset easier.

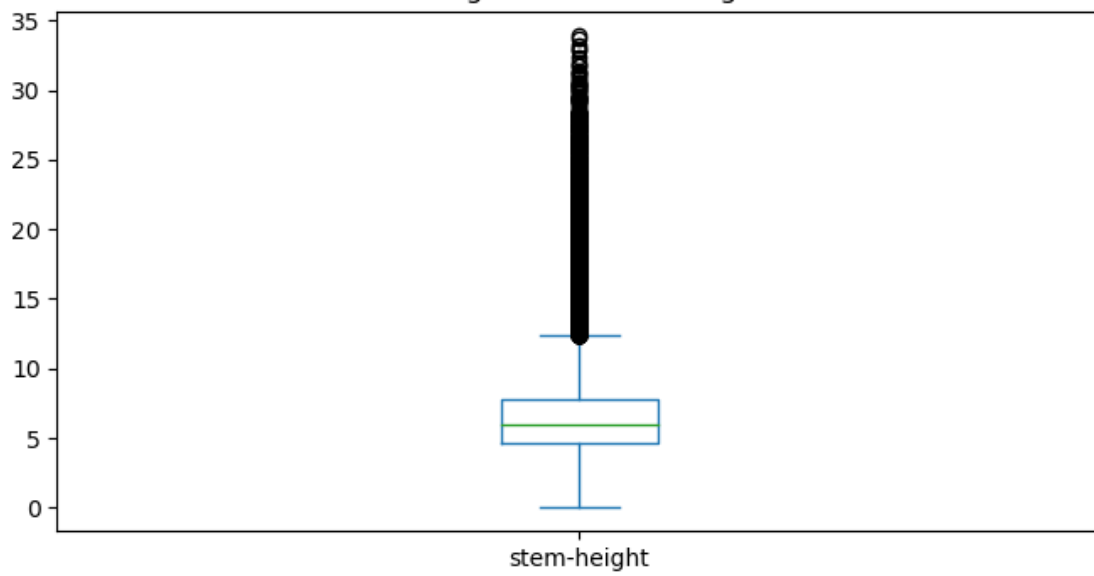
3.5 Outlier detection

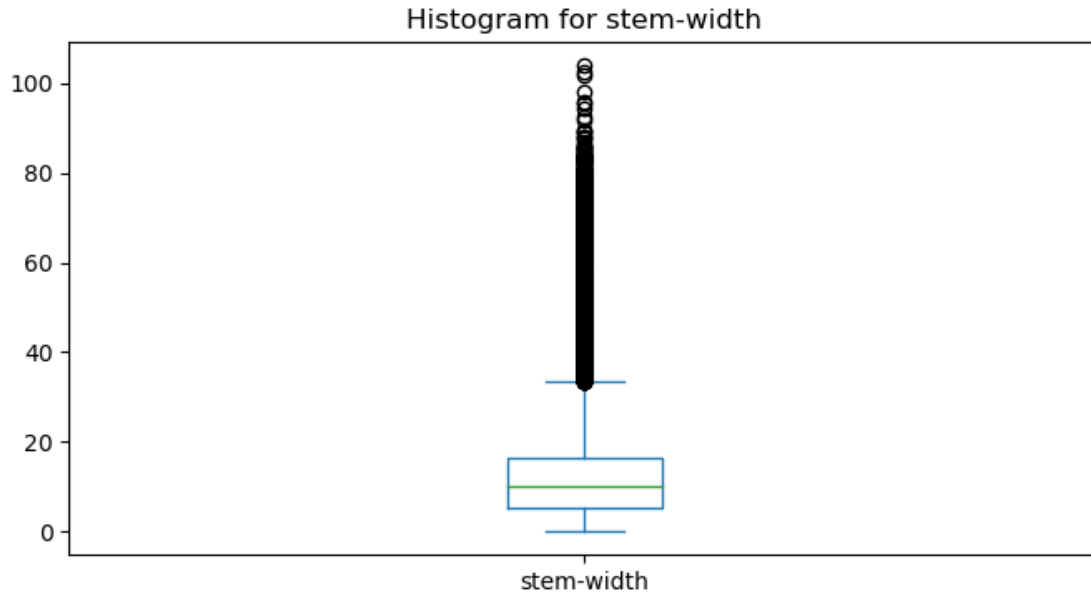
```
[ ]: for column in numerical_columns:
    fig, ax = plt.subplots(figsize=(8, 4))
    df[column].plot(kind='box', title=f"Histogram for {column}", ax=ax)
    plt.show()
```

Histogram for cap-diameter



Histogram for stem-height





By the box plot, we can see there are a lot of outlier in the positive side for all numerical columns.

```
[ ]: outliers_dict = dict()

for column in numerical_columns:
    # Calculate the first quartile (Q1) and third quartile (Q3)
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)

    # Calculate the interquartile range (IQR)
    IQR = Q3 - Q1

    # Define the lower and upper bounds for outliers
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Identify outliers
    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)].loc[:,column]

    outliers_dict[column] = outliers.count()
```

```
[ ]: outliers_dict
```

```
[ ]: {'cap-diameter': 2400, 'stem-height': 3169, 'stem-width': 1967}
```

Since the amount of outliers is small compare with the total rows, I will not remove any outliers.

And I will use ensemble model. They are able to handle the outliers.

4 Model Training and Evaluation

The models I will use is logistic regression, Decision Tree, and KNN * Logistic Regression: simple model, suitable for binary classification especially when the dataset have linear pattern * Decision Tree: model that is capable for regression and classification. This model is able to find the important features easily, and capture the non-linear pattern which logistic regression cannot. * KNN: simple model that can capture nonlinear pattern, but it require to compute the distance for all data points which would are complex when the dataset is large.

All of these models are easy to understand and easy to implement, and these models are representor of Gradient-based, Tree-based and Distance-based models.

```
[ ]: from sklearn.model_selection import train_test_split
      train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.
      ↪ 2, random_state=42)
```

4.1 Logistic Regression

```
[ ]: # import numpy as np
      # import pandas as pd

      # class MyLogisticRegression:

      #     def __init__(self, epochs = 10000, threshold=1e-3,
      #                   regularization=None, alpha=0.01) -> None:

      #         # Initialize parameters and flags for the model.
      #         self.epochs = epochs
      #         self.threshold = threshold
      #         self.regularization = regularization
      #         self.alpha = alpha

      #     def train(self, X, y, batch_size=64, lr=1e-3, seed=11, verbose=False):
      #         """
      #         Train the model using stochastic gradient descent.
      #         """
      #         # Set seed for reproducibility.
      #         np.random.seed(seed)

      #         # Define the unique classes and their corresponding indices.
      #         self.classes = np.unique(y)
      #         self.class_labels = {c: i for i, c in enumerate(self.classes)}

      #         # Add bias term to the features.
```



```

#         X = self.add_bias(X)

#         # Convert labels into one-hot encoded format.
#         y = self.one_hot(y)

#         # Initialize weights matrix with zeros.
#         self.loss = []
#         self.weights = np.zeros(shape=(len(self.classes), X.shape[1]))

#         # Start the training process.
#         self.fit_data(X, y, batch_size, lr, verbose)
#         return self

def fit_data(self, X, y, batch_size, lr, verbose):
    """
    Fit the data using stochastic gradient descent.
    """
    i = 0
    while (not self.epochs or i < self.epochs):
        # Compute and store the cross-entropy loss.
        self.loss.append(self.cross_entropy(y, self.predict_(X)))

        # Randomly select a batch of data.
        idx = np.random.choice(X.shape[0], batch_size)
        X_batch, y_batch = X[idx], y[idx]

        # Calculate the error between predicted and true values.
        error = y_batch - self.predict_(X_batch)

        # Update the weights based on the error and learning rate.
        update = lr * np.dot(error.T, X_batch)

        # Apply regularization if specified.
        if self.regularization == 'Ridge':
            update += self.alpha * self.weights
        elif self.regularization == 'Lasso':
            update += self.alpha * np.sign(self.weights)
        elif self.regularization == 'Elastic Net':
            update_w += self.alpha * (self.weights + np.sign(self.
↪weights))

        self.weights += update

        # Stop training if updates are smaller than a threshold.
        if np.abs(update).max() < self.threshold:

```

```

#             break

#             # Print training accuracy every 1000 iterations if verbose is
#             ↪ True.
#             if i % 1000 == 0 and verbose:
#                 print(' Training Accuracy at {} iterations is {}'.format(i,
#             ↪ self.evaluate_(X, y)))
#                 i += 1

#     def predict(self, X):
#         return self.predict_(self.add_bias(X))

#     def predict_(self, X):
#         pre_vals = np.dot(X, self.weights.T).reshape(-1, len(self.classes))
#         return self.softmax(pre_vals)

#     def softmax(self, z):
#         return np.exp(z) / np.sum(np.exp(z), axis=1).reshape(-1,1)

#     def predict_classes(self, X):
#         self.probs_ = self.predict(X)
#         return np.vectorize(lambda c: self.classes[c])(np.argmax(self.probs_,
#             ↪ axis=1))

#     def add_bias(self, X):
#         return np.insert(X, 0, 1, axis=1)

#     def one_hot(self, y):
#         return np.eye(len(self.classes))[np.vectorize(lambda c: self.
#             ↪ class_labels[c])(y).reshape(-1)]

#     def score(self, X, y):
#         '''
#         Accuracy metric
#         '''
#         return round(np.mean(self.predict_classes(X).reshape(-1,1) == y),3)

#     def evaluate_(self, X, y):
#         return np.mean(np.argmax(self.predict_(X), axis=1) == np.argmax(y,
#             ↪ axis=1))

#     def cross_entropy(self, y, probs):
#         return -1 * np.mean(y * np.log(probs))

#     def confusion_matrix(self, actual, predicted, norm=False):
#         """

```

```

#         Compute the confusion matrix for the given actual and predicted
#         ↪ outputs.

#         Args:
#         - actual (array-like): Actual outputs (ground truth).
#         - predicted (array-like): Predicted outputs from the model.

#         Returns:
#         - matrix (np.ndarray): N x N confusion matrix, where N is the number
#         ↪ of unique classes.
#         """

#         # Create an empty matrix
#         matrix = np.zeros((len(self.classes), len(self.classes)), dtype=float)

#         # Fill the matrix
#         for i, true_class in enumerate(self.classes):
#             for j, pred_class in enumerate(self.classes):
#                 matrix[i, j] = np.sum((actual == true_class) & (predicted ==
#                 ↪ pred_class))

#         if norm:
#             for i in range(len(self.classes)):
#                 total = np.sum(matrix[i])
#                 for j in range(len(self.classes)):
#                     matrix[i, j] = round(matrix[i, j] / total, 2)

#         matrix_df = pd.DataFrame(matrix, index=self.classes, columns=self.
#         ↪ classes)
#         return matrix_df

```

```

[ ]: from LogisticRegression import MyLogisticRegression
lm = MyLogisticRegression()
lm.train(train_X.values, train_y.values)

```

```

[ ]: <LogisticRegression.MyLogisticRegression at 0x133453650>

```

4.1.1 Training set evaluation

```

[ ]: pred_train_y = lm.predict_classes(train_X.values)
lm.confusion_matrix(train_y, pred_train_y)

```

```

[ ]:
      e      p
e 20183.0 1624.0
p  7378.0 19670.0

```

logistic regression can classify e much better than p, but the overall performance is great. We treat e as positive and p as negative, then if we misclassify e to p, we call it type 1 error(false positive). And misclassify p as e, we call it type 2 error(false negative). Type 2 is bad for this case, because people eat the poisonous mushrooms which were classified as edible. Therefore, we should increase the overall accuracy or decrease the type 2 error.

```
[ ]: print(f'accuracy: {np.mean(pred_train_y == train_y)}')
```

accuracy: 0.8157404564527684

Logistic regression perform great in training set, acquire 81.5% accuracy.

4.1.2 Test set evaluation

```
[ ]: pred_test_y = lm.predict_classes(test_X)
lm.confusion_matrix(test_y,pred_test_y)
```

```
[ ]:
      e      p
e 4934.0  440.0
p 1865.0 4975.0
```

```
[ ]: print(f'accuracy: {np.mean(pred_test_y == test_y)}')
```

accuracy: 0.8112821352546259

The logistic model seems perform about the same in test dataset. It acquires 81.1% accuracy which is just 0.4% lower than the accuracy in test dataset.

4.1.3 Important features

```
[ ]: pd.set_option('display.max_columns', None)
pd.DataFrame(lm.weights,index=[0,1],columns=X.columns.insert(0,'bias'))
```

```
[ ]:
      bias  cap-diameter  stem-height  stem-width  cap-shape_b  cap-shape_c  \
0 -1.465632      0.165223      0.0796      0.092808     -1.341672     -0.191079
1  1.465632     -0.165223     -0.0796     -0.092808      1.341672      0.191079

      cap-shape_f  cap-shape_o  cap-shape_p  cap-shape_s  cap-shape_x  \
0      0.104998     -0.926183      0.137476      0.482723      0.268105
1     -0.104998      0.926183     -0.137476     -0.482723     -0.268105

      cap-surface_?  cap-surface_d  cap-surface_e  cap-surface_g  cap-surface_h  \
0      0.229483      0.241876     -0.66786      0.811854      0.623892
1     -0.229483     -0.241876      0.66786     -0.811854     -0.623892

      cap-surface_i  cap-surface_k  cap-surface_l  cap-surface_s  cap-surface_t  \
0     -1.622021     -3.011944      1.189832      0.574294     -0.686148
1      1.622021      3.011944     -1.189832     -0.574294      0.686148
```

	cap-surface_w	cap-surface_y	cap-color_b	cap-color_e	cap-color_g	\
0	0.068476	0.782634	1.066569	-1.446941	0.35109	
1	-0.068476	-0.782634	-1.066569	1.446941	-0.35109	

	cap-color_k	cap-color_l	cap-color_n	cap-color_o	cap-color_p	\
0	-0.618687	1.166477	0.849755	-0.035764	-0.606174	
1	0.618687	-1.166477	-0.849755	0.035764	0.606174	

	cap-color_r	cap-color_u	cap-color_w	cap-color_y	does-bruise-or-bleed_f	\
0	-2.086839	-0.480523	0.090853	0.284552	-0.827921	
1	2.086839	0.480523	-0.090853	-0.284552	0.827921	

	does-bruise-or-bleed_t	gill-attachment_?	gill-attachment_a	\
0	-0.637711	-0.801695	-0.893634	
1	0.637711	0.801695	0.893634	

	gill-attachment_d	gill-attachment_e	gill-attachment_f	gill-attachment_p	\
0	-2.158874	1.088682	-0.047301	2.981584	
1	2.158874	-1.088682	0.047301	-2.981584	

	gill-attachment_s	gill-attachment_x	gill-spacing_?	gill-spacing_c	\
0	-0.306381	-1.328013	-1.006156	-0.38982	
1	0.306381	1.328013	1.006156	0.38982	

	gill-spacing_d	gill-spacing_f	gill-color_b	gill-color_e	gill-color_f	\
0	-0.022356	-0.047301	1.247254	-0.938048	-0.047301	
1	0.022356	0.047301	-1.247254	0.938048	0.047301	

	gill-color_g	gill-color_k	gill-color_n	gill-color_o	gill-color_p	\
0	0.14095	-0.856506	-0.848034	0.174075	0.24918	
1	-0.14095	0.856506	0.848034	-0.174075	-0.24918	

	gill-color_r	gill-color_u	gill-color_w	gill-color_y	stem-root_?	\
0	0.215163	0.012258	0.46076	-1.275383	2.445175	
1	-0.215163	-0.012258	-0.46076	1.275383	-2.445175	

	stem-root_b	stem-root_c	stem-root_f	stem-root_r	stem-root_s	\
0	2.774851	-3.522699	-0.151454	-2.844927	-0.166577	
1	-2.774851	3.522699	0.151454	2.844927	0.166577	

	stem-surface_?	stem-surface_f	stem-surface_g	stem-surface_h	\
0	1.212086	-0.151454	-3.119523	-1.049839	
1	-1.212086	0.151454	3.119523	1.049839	

	stem-surface_i	stem-surface_k	stem-surface_s	stem-surface_t	\
0	-0.006051	-0.816623	1.511848	2.198119	
1	0.006051	0.816623	-1.511848	-2.198119	

	stem-surface_y	stem-color_b	stem-color_e	stem-color_f	stem-color_g	\
0	-1.244194	0.460045	-1.315729	-0.151454	1.145737	
1	1.244194	-0.460045	1.315729	0.151454	-1.145737	

	stem-color_k	stem-color_l	stem-color_n	stem-color_o	stem-color_p	\
0	-0.975333	0.560633	-0.059773	0.901764	-1.59306	
1	0.975333	-0.560633	0.059773	-0.901764	1.59306	

	stem-color_r	stem-color_u	stem-color_w	stem-color_y	veil-type_?	\
0	-0.379744	-0.795127	1.315065	-0.578656	1.196461	
1	0.379744	0.795127	-1.315065	0.578656	-1.196461	

	veil-type_u	veil-color_?	veil-color_e	veil-color_k	veil-color_n	\
0	-2.662093	-0.688759	-0.540438	-0.184157	-1.108955	
1	2.662093	0.688759	0.540438	0.184157	1.108955	

	veil-color_u	veil-color_w	veil-color_y	has-ring_f	has-ring_t	\
0	-1.147823	0.706619	1.497881	0.278252	-1.743884	
1	1.147823	-0.706619	-1.497881	-0.278252	1.743884	

	ring-type_?	ring-type_e	ring-type_f	ring-type_g	ring-type_l	\
0	0.88374	-1.047384	-1.160445	0.750416	0.387372	
1	-0.88374	1.047384	1.160445	-0.750416	-0.387372	

	ring-type_m	ring-type_p	ring-type_r	ring-type_z	spore-print-color_?	\
0	1.548294	-0.894767	1.08723	-3.020089	-0.029702	
1	-1.548294	0.894767	-1.08723	3.020089	0.029702	

	spore-print-color_g	spore-print-color_k	spore-print-color_n	\
0	1.168732	-1.231195	-1.083105	
1	-1.168732	1.231195	1.083105	

	spore-print-color_p	spore-print-color_r	spore-print-color_u	\
0	-1.023482	-0.08949	-0.186547	
1	1.023482	0.08949	0.186547	

	spore-print-color_w	habitat_d	habitat_g	habitat_h	habitat_l	habitat_m	\
0	1.009156	-0.585411	-0.645654	-0.198758	-0.226998	0.24723	
1	-1.009156	0.585411	0.645654	0.198758	0.226998	-0.24723	

	habitat_p	habitat_u	habitat_w	season_a	season_s	season_u	season_w
0	-0.693242	0.133146	0.504056	-0.962381	0.229055	-1.014429	0.282124
1	0.693242	-0.133146	-0.504056	0.962381	-0.229055	1.014429	-0.282124

- cap-surface_g: 0.81
- cap-color_l: 1.16

- ring-type_m: 1.54

These are the important features. The important feature in logistic regression would have higher weight

4.2 Decision Tree

```
[ ]: # import numpy as np
# import pandas as pd

# class Node():
#     def __init__(self, feature_index=None, threshold=None, left=None,
#         ↪right=None, var_red=None, info_gain=None, value=None):
#         ''' constructor '''

#         # for decision node
#         self.feature_index = feature_index
#         self.threshold = threshold # threshold for splitting data
#         self.left = left
#         self.right = right
#         self.var_red = var_red # index for regressor, choose the highest
#         ↪feature to split
#         self.info_gain = info_gain # index for classifier, choose the highest
#         ↪to split

#         # for leaf node
#         self.value = value

# # =====Regression=====
# class DecisionTreeRegressor():
#     def __init__(self, min_samples_split=2, max_depth=2):
#         ''' constructor '''

#         # initialize the root of the tree
#         self.root = None

#         # stopping conditions
#         self.min_samples_split = min_samples_split # min elements need to
#         ↪split
#         self.max_depth = max_depth # max depth for tree

#     def build_tree(self, dataset, curr_depth=0):
#         ''' recursive function to build the tree '''

#         X, Y = dataset[:, :-1], dataset[:, -1]
#         num_samples, num_features = np.shape(X)
```

```

#         best_split = {}
#         # split until stopping conditions are met
#         if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
#             # find the best split
#             best_split = self.get_best_split(dataset, num_samples,
#         ↪num_features)
#             # check if information gain is positive
#             if best_split["var_red"]>0:
#                 # recur left
#                 left_subtree = self.build_tree(best_split["dataset_left"],
#         ↪curr_depth+1)
#                 # recur right
#                 right_subtree = self.build_tree(best_split["dataset_right"],
#         ↪curr_depth+1)
#             # return decision node
#             return Node(best_split["feature_index"],
#         ↪best_split["threshold"],
#                 left_subtree, right_subtree,
#         ↪best_split["var_red"])

#         # compute leaf node
#         leaf_value = self.calculate_leaf_value(Y)
#         # return leaf node
#         return Node(value=leaf_value)

#     def get_best_split(self, dataset, num_samples, num_features):
#         '''
#         #         function to find the best split
#
#         Returns:
#             Dict: A dict that contains information for best split
#         '''
#
#         # dictionary to store the best split
#         best_split = {}
#         max_var_red = -float("inf")
#         # loop over all the features
#         for feature_index in range(num_features):
#             feature_values = dataset[:, feature_index]
#             possible_thresholds = np.unique(feature_values)
#             # loop over all the feature values present in the data
#             for threshold in possible_thresholds:
#                 # get current split
#                 dataset_left, dataset_right = self.split(dataset,
#         ↪feature_index, threshold)
#                 # check if childs are not null

```



```

#         if len(dataset_left)>0 and len(dataset_right)>0:
#             y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],
#             ↪dataset_right[:, -1]
#             # compute information gain
#             curr_var_red = self.variance_reduction(y, left_y, right_y)
#             # update the best split if needed
#             # if curr is higher, update everything
#             if curr_var_red>max_var_red:
#                 best_split["feature_index"] = feature_index
#                 best_split["threshold"] = threshold
#                 best_split["dataset_left"] = dataset_left
#                 best_split["dataset_right"] = dataset_right
#                 best_split["var_red"] = curr_var_red
#                 max_var_red = curr_var_red
#
#         # return best split
#         return best_split
#
#     def split(self, dataset, feature_index, threshold):
#         ''' function to split the data using a feature and threshold
#
#         Returns:
#             Tuple: dataset_left, dataset_right
#         '''
#
#         dataset_left = np.array([row for row in dataset if
#             ↪row[feature_index]<=threshold])
#         dataset_right = np.array([row for row in dataset if
#             ↪row[feature_index]>threshold])
#         return dataset_left, dataset_right
#
#     def variance_reduction(self, parent, l_child, r_child):
#         ''' function to compute variance reduction
#
#         Returns:
#             float: measure the information gain for a certain split
#         '''
#
#         weight_l = len(l_child) / len(parent)
#         weight_r = len(r_child) / len(parent)
#         reduction = np.var(parent) - (weight_l * np.var(l_child) + weight_r *
#             ↪np.var(r_child))
#         return reduction
#
#     def calculate_leaf_value(self, Y):
#         ''' function to compute leaf node using mean

```

```

#         Returns:
#         float: mean value for given node value
#         '''

#         val = np.mean(Y)
#         return val

#     def print_tree(self, tree=None, indent=" "):
#         ''' function to print the tree '''

#         if not tree:
#             tree = self.root

#         if tree.value is not None:
#             print(tree.value)

#         else:
#             print("X_"+str(tree.feature_index), "<=", tree.threshold, "?",
↳ tree.var_red)
#             print("%sleft:" % (indent), end="")
#             self.print_tree(tree.left, indent + indent)
#             print("%sright:" % (indent), end="")
#             self.print_tree(tree.right, indent + indent)

#     def pretty_print(self, node=None, depth=0, prefix="Root: "):
#         '''Print tree in pretty format using post order traversal'''

#         if node is None:
#             node = self.root

#         indent = " " * depth * 4

#         if node.value is not None:
#             print(f"{indent}{prefix}Leaf => Value: {round(node.value,2)} |
↳ depth: {depth}")
#         else:
#             self.pretty_print(node.right, depth + 1, prefix="R--> ")
#             print(f"{indent}{prefix} X_{node.feature_index} <= {round(node.
↳ threshold,2)} | depth: {depth}")
#             self.pretty_print(node.left, depth + 1, prefix="L--> ")

#     def fit(self, X, Y):
#         ''' function to train the tree '''

#         dataset = np.concatenate((X, Y), axis=1)
#         self.root = self.build_tree(dataset)

```

```

#     def make_prediction(self, x, tree):
#         ''' function to predict new dataset '''

#         if tree.value!=None: return tree.value
#         feature_val = x[tree.feature_index]
#         if feature_val<=tree.threshold:
#             return self.make_prediction(x, tree.left)
#         else:
#             return self.make_prediction(x, tree.right)

#     def predict(self, X):
#         ''' function to predict a single data point '''

#         predictions = [self.make_prediction(x, self.root) for x in X]
#         return predictions

#     def mse(self,y_true, y_pred,verbose=False):
#         ''' Calculate Mean Square Error'''

#         res = np.mean((y_true - y_pred)**2)
#         if verbose:
#             print(f"MSE: {res}")
#         return res

#     def r2(self,y_true,y_pred,verbose=False):
#         ''' Calculate r2 score for regressor, range form -inf to 1, higher
↳ means better
#         '''
#         y_mean = np.mean(y_true)

#         # Calculate total sum of squares
#         SST = np.sum((y_true - y_mean) ** 2)

#         # Calculate residual sum of squares
#         SSR = np.sum((y_true - y_pred) ** 2)

#         # Calculate R2 score
#         r2 = 1 - (SSR / SST)

#         return r2

# # =====Regression=====

```

```

# #
# =====Classification=====
# class DecisionTreeClassifier():
#     def __init__(self, min_samples_split=2, max_depth=2):
#         ''' constructor '''

#         # initialize the root of the tree
#         self.root = None

#         # stopping conditions
#         self.min_samples_split = min_samples_split
#         self.max_depth = max_depth

#     def build_tree(self, dataset, curr_depth=0):
#         ''' recursive function to build the tree '''

#         X, Y = dataset[:, :-1], dataset[:, -1]
#         num_samples, num_features = np.shape(X)

#         # split until stopping conditions are met
#         if num_samples >= self.min_samples_split and curr_depth <= self.max_depth:
#             # find the best split
#             best_split = self.get_best_split(dataset, num_samples,
#         num_features)

#             # check if information gain is positive
#             if best_split["info_gain"] > 0:
#                 # recur left
#                 left_subtree = self.build_tree(best_split["dataset_left"],
#         curr_depth+1)

#                 # recur right
#                 right_subtree = self.build_tree(best_split["dataset_right"],
#         curr_depth+1)

#             # return decision node
#             return Node(best_split["feature_index"],
#         best_split["threshold"],
#         left_subtree,
#         right_subtree, info_gain=best_split["info_gain"])

#         # compute leaf node
#         leaf_value = self.calculate_leaf_value(Y)
#         # return leaf node
#         return Node(value=leaf_value)

#     def get_best_split(self, dataset, num_samples, num_features):
#         ''' function to find the best split '''

```

```

#         # dictionary to store the best split
#         best_split = {}
#         max_info_gain = -float("inf")

#         # loop over all the features
#         for feature_index in range(num_features):
#             feature_values = dataset[:, feature_index]
#             possible_thresholds = np.unique(feature_values)
#             # loop over all the feature values present in the data
#             for threshold in possible_thresholds:
#                 # get current split
#                 dataset_left, dataset_right = self.split(dataset,
# ↪feature_index, threshold)
#                 # check if childs are not null
#                 if len(dataset_left)>0 and len(dataset_right)>0:
#                     y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],
# ↪dataset_right[:, -1]
#                     # compute information gain
#                     curr_info_gain = self.information_gain(y, left_y,
# ↪right_y, "gini")
#                     # update the best split if needed
#                     if curr_info_gain>max_info_gain:
#                         best_split["feature_index"] = feature_index
#                         best_split["threshold"] = threshold
#                         best_split["dataset_left"] = dataset_left
#                         best_split["dataset_right"] = dataset_right
#                         best_split["info_gain"] = curr_info_gain
#                         max_info_gain = curr_info_gain

#         # return best split
#         return best_split

#     def split(self, dataset, feature_index, threshold):
#         ''' function to split the data '''

#         dataset_left = np.array([row for row in dataset if
# ↪row[feature_index]<=threshold])
#         dataset_right = np.array([row for row in dataset if
# ↪row[feature_index]>threshold])
#         return dataset_left, dataset_right

#     def information_gain(self, parent, l_child, r_child, mode="entropy"):
#         ''' function to compute information gain '''

#         weight_l = len(l_child) / len(parent)
#         weight_r = len(r_child) / len(parent)
#         if mode=="gini":

```

```

#         gain = self.gini_index(parent) - (weight_l*self.
↳gini_index(l_child) + weight_r*self.gini_index(r_child))
#         else:
#         gain = self.entropy(parent) - (weight_l*self.entropy(l_child) +
↳weight_r*self.entropy(r_child))
#         return gain

#     def entropy(self, y):
#         ''' function to compute entropy '''

#         class_labels = np.unique(y)
#         entropy = 0
#         for cls in class_labels:
#             p_cls = len(y[y == cls]) / len(y)
#             entropy += -p_cls * np.log2(p_cls)
#         return entropy

#     def gini_index(self, y):
#         ''' function to compute gini index '''

#         class_labels = np.unique(y)
#         gini = 0
#         for cls in class_labels:
#             p_cls = len(y[y == cls]) / len(y)
#             gini += p_cls**2
#         return 1 - gini

#     def calculate_leaf_value(self, Y):
#         ''' function to compute leaf node '''

#         Y = list(Y)
#         # return the maximum count element
#         return max(Y, key=Y.count)

#     def print_tree(self, tree=None, indent=" "):
#         ''' function to print the tree '''

#         if not tree:
#             tree = self.root

#         if tree.value is not None:
#             print(tree.value)

#         else:
#             print("X_"+str(tree.feature_index), "<=", tree.threshold, "?",
↳tree.info_gain)
#             print("%sleft:" % (indent), end="")

```

```

#         self.print_tree(tree.left, indent + indent)
#         print("%sright:" % (indent), end="")
#         self.print_tree(tree.right, indent + indent)

#     def pretty_print(self, node=None, depth=0, prefix="Root: "):
#         '''Print tree in pretty format using post order traversal'''

#         if node is None:
#             node = self.root

#         indent = " " * depth * 4

#         # print leaves node
#         if node.value is not None:
#             print(f"{indent}{prefix}Leaf => Value: {node.value} | depth: {depth}")
#         # print feature node
#         else:
#             self.pretty_print(node.right, depth + 1, prefix="R--> ")
#             print(f"{indent}{prefix} X_{node.feature_index} <= {node.threshold} | depth: {depth}")
#             self.pretty_print(node.left, depth + 1, prefix="L--> ")

#     def fit(self, X, Y):
#         ''' function to train the tree '''

#         dataset = np.concatenate((X, Y), axis=1)
#         self.root = self.build_tree(dataset)

#     def predict(self, X):
#         ''' function to predict new dataset

#         Returns:
#             List: list contains predictions
#         '''

#         predictions = [self.make_prediction(x, self.root) for x in X]
#         return predictions

#     def make_prediction(self, x, tree):
#         ''' function to predict a single data point '''

#         # if the node is leaf node, return the value
#         if tree.value is not None: return tree.value
#         # otherwise travel to left or right node by the threshold
#         feature_val = x[tree.feature_index]
#         if feature_val <= tree.threshold:

```

```

#         return self.make_prediction(x, tree.left)
#     else:
#         return self.make_prediction(x, tree.right)

#     def accuracy(self, y_true, y_pred):
#         ''' Calculate the accuracy '''

#         count = 0
#         N = len(y_true)

#         for i in range(N):
#             if y_true[i] == y_pred[i]:
#                 count += 1

#         return count / N
# #_
↪=====Classification=====

```

```
[ ]: from DST import DecisionTreeClassifier
```

```
[ ]: dt = DecisionTreeClassifier(min_samples_split=5,max_depth=10)
dt.fit(train_X.values,train_y.values.reshape(-1,1))
```

4.2.1 Training set

```
[ ]: pred_y = dt.predict(train_X.values)
```

```
[ ]: np.mean(pred_y == train_y)
```

```
[ ]: 0.8666052604646403
```

4.2.2 test set

```
[ ]: pred_y = dt.predict(test_X.values)
np.mean(pred_y == test_y)
```

```
[ ]: 0.8648272474209923
```

4.2.3 important features

```
[ ]: dt.pretty_print()
```

```

R--> Leaf => Value: p | depth: 2
R--> X_68 <= 0.0 | depth: 1
      R--> Leaf => Value: p | depth: 3
      L--> X_107 <= 0.0 | depth: 2

```



```

R--> Leaf => Value: e | depth: 4
L--> X_72 <= 0.0 | depth: 3
      R--> Leaf => Value: p | depth: 5
      L--> X_64 <= 0.0 | depth: 4
            R--> Leaf =>
Value: e | depth: 8
            R--> X_37 <= 0.0 |
depth: 7
R--> Leaf => Value: p | depth: 10
            R-->
X_90 <= 0.0 | depth: 9
L--> Leaf => Value: e | depth: 10
            L--> X_0 <=
10.06 | depth: 8
      R--> Leaf => Value: p | depth: 11
R--> X_33 <= 0.0 | depth: 10
      L--> Leaf => Value: e | depth: 11
            L-->
X_2 <= 9.26 | depth: 9
L--> Leaf => Value: p | depth: 10
            R--> X_59 <= 0.0 | depth: 6
            R--> Leaf =>
Value: p | depth: 8
            L--> X_87 <= 0.0 |
depth: 7
            R-->
Leaf => Value: p | depth: 9
            L--> X_24 <=
0.0 | depth: 8
      R--> Leaf => Value: p | depth: 11
R--> X_74 <= 0.0 | depth: 10
      L--> Leaf => Value: e | depth: 11
            L-->
X_0 <= 3.46 | depth: 9
      R--> Leaf => Value: p | depth: 11
L--> X_32 <= 0.0 | depth: 10
      L--> Leaf => Value: e | depth: 11
            L--> X_18 <= 0.0 | depth: 5
            R--> Leaf =>
Value: e | depth: 8
            R--> X_56 <= 0.0 |
depth: 7
            L--> Leaf =>
Value: p | depth: 8
            L--> X_30 <= 0.0 | depth: 6
R--> Leaf => Value: e | depth: 10
            R-->
X_11 <= 0.0 | depth: 9

```

```

L--> Leaf => Value: p | depth: 10
                                R--> X_108 <=
0.0 | depth: 8
                                L-->
Leaf => Value: e | depth: 9
                                L--> X_46 <= 0.0 |
depth: 7
                                R-->
Leaf => Value: p | depth: 9
                                L--> X_83 <=
0.0 | depth: 8
R--> Leaf => Value: p | depth: 10
                                L-->
X_62 <= 0.0 | depth: 9
    R--> Leaf => Value: e | depth: 11
L--> X_41 <= 0.0 | depth: 10
    L--> Leaf => Value: e | depth: 11
Root: X_2 <= 8.22 | depth: 0
    R--> Leaf => Value: p | depth: 4
    R--> X_17 <= 0.0 | depth: 3
        R--> Leaf => Value: p | depth: 5
        L--> X_11 <= 0.0 | depth: 4
            R--> Leaf => Value: e | depth: 6
            L--> X_2 <= 1.79 | depth: 5
                L--> Leaf => Value: p | depth: 6
R--> X_1 <= 3.69 | depth: 2
    R--> Leaf => Value: e | depth: 4
    L--> X_2 <= 3.01 | depth: 3
        R--> Leaf => Value: e | depth: 5
        L--> X_3 <= 0.0 | depth: 4
            R--> Leaf => Value: e | depth: 6
            L--> X_5 <= 0.0 | depth: 5
                R--> Leaf => Value: e |
depth: 7
                                L--> X_21 <= 0.0 | depth: 6
                                L--> Leaf => Value: p |
depth: 7
    L--> X_46 <= 0.0 | depth: 1
        R--> Leaf => Value: e | depth: 5
        R--> X_17 <= 0.0 | depth: 4
            R--> Leaf => Value: e | depth: 6
            L--> X_10 <= 0.0 | depth: 5
                L--> Leaf => Value: p | depth: 6
R--> X_117 <= 0.0 | depth: 3
    R--> Leaf => Value: p | depth: 6
    R--> X_4 <= 0.0 | depth: 5
        R--> Leaf => Value: p |
depth: 7

```

```

R--> Leaf => Value: e | depth: 10
L--> X_65 <= 0.0 | depth: 6
R-->
X_21 <= 0.0 | depth: 9
R--> Leaf => Value: e | depth: 11
L--> X_17 <= 0.0 | depth: 10
L--> Leaf => Value: p | depth: 11
R--> X_44 <=
0.0 | depth: 8
R--> Leaf => Value: p | depth: 10
L-->
X_37 <= 0.0 | depth: 9
L--> Leaf => Value: e | depth: 10
L--> X_58 <= 0.0 |
depth: 7
R--> Leaf => Value: e | depth: 11
R--> X_0 <= 2.84 | depth: 10
L--> Leaf => Value: p | depth: 11
R-->
X_55 <= 0.0 | depth: 9
R--> Leaf => Value: p | depth: 11
L--> X_70 <= 0.0 | depth: 10
L--> Leaf => Value: e | depth: 11
L--> X_2 <=
0.93 | depth: 8
L-->
Leaf => Value: p | depth: 9
L--> X_97 <= 0.0 | depth: 4
R--> Leaf => Value: e | depth: 6
L--> X_1 <= 7.23 | depth: 5
L--> Leaf => Value: p | depth: 6
L--> X_86 <= 0.0 | depth: 2
R--> Leaf => Value: e | depth: 5
R--> X_13 <= 0.0 | depth: 4
L--> Leaf => Value: p | depth: 5
L--> X_4 <= 0.0 | depth: 3
R--> Leaf => Value: e | depth: 6
R--> X_106 <= 0.0 | depth: 5
R--> Leaf =>
Value: e | depth: 8
R--> X_11 <= 0.0 |
depth: 7
R-->
Leaf => Value: e | depth: 9
L--> X_9 <= 0.0
| depth: 8
L-->
Leaf => Value: p | depth: 9

```

```

R--> Leaf => Value: e | depth: 10
L--> X_55 <= 0.0 | depth: 6
R-->
X_43 <= 0.0 | depth: 9
L--> Leaf => Value: p | depth: 10
R--> X_58 <=
0.0 | depth: 8
R--> Leaf => Value: e | depth: 10
L-->
X_34 <= 0.0 | depth: 9
L--> Leaf => Value: p | depth: 10
L--> X_10 <= 0.0 |
depth: 7
R--> Leaf => Value: p | depth: 11
R--> X_8 <= 0.0 | depth: 10
L--> Leaf => Value: e | depth: 11
R-->
X_1 <= 4.08 | depth: 9
R--> Leaf => Value: e | depth: 11
L--> X_117 <= 0.0 | depth: 10
L--> Leaf => Value: p | depth: 11
L--> X_54 <=
0.0 | depth: 8
R--> Leaf => Value: p | depth: 11
R--> X_15 <= 0.0 | depth: 10
L--> Leaf => Value: e | depth: 11
L-->
X_49 <= 0.0 | depth: 9
R--> Leaf => Value: e | depth: 11
L--> X_22 <= 0.0 | depth: 10
L--> Leaf => Value: p | depth: 11
L--> X_2 <= 5.24 | depth: 4
R--> Leaf =>
Value: e | depth: 8
R--> X_9 <= 0.0 |
depth: 7
L--> Leaf =>
Value: p | depth: 8
R--> X_73 <= 0.0 | depth: 6
R-->
Leaf => Value: e | depth: 9
R--> X_43 <=
0.0 | depth: 8
R--> Leaf => Value: p | depth: 10
L-->
X_1 <= 2.95 | depth: 9
L--> Leaf => Value: e | depth: 10
L--> X_14 <= 0.0 |

```

```

depth: 7
R--> Leaf => Value: e | depth: 10

X_10 <= 0.0 | depth: 9
L--> Leaf => Value: p | depth: 10

| depth: 8

Leaf => Value: p | depth: 9

L--> X_34 <= 0.0 | depth: 5
R--> Leaf =>

Value: e | depth: 8

R--> X_11 <= 0.0 |

depth: 7

L--> Leaf =>

Value: p | depth: 8

L--> X_0 <= 1.4 | depth: 6
L--> Leaf => Value: e |

depth: 7

```

The first 3 split is 2,46,68. Since decision tree will find the use best split in each decision node, the most important features will be in the first coupes splits.

```
[ ]: X.iloc[:,[2,26,68]].columns
```

```
[ ]: Index(['stem-width', 'cap-color_l', 'stem-surface_g'], dtype='object')
```

‘stem-width’, ‘cap-color_l’, ‘stem-surface_g’ are the important features.

4.3 KNN

```
[ ]: # import numpy as np
# from collections import Counter
# class KNN:

#     def __init__(self,k=3):
#         self.k = k

#     def fit(self,X,y):
#         self.X_train = X
#         self.y_train = y

#     def predict(self,X):
#         pred_y = [self._predict(x) for x in X]
#         return np.array(pred_y)

#     def _predict(self,x):

```

```
#         distances = np.linalg.norm(self.X_train - x, axis=1)

#         # Find the indices of the k nearest neighbors
#         k_index = np.argpartition(distances, self.k)[:self.k]

#         # Retrieve the labels of the k nearest neighbors
#         k_nearest_labels = self.y_train[k_index]

#         return list(Counter(k_nearest_labels).keys())[0]
```

```
[ ]: from pca import PCA
      from KNN import KNN
      pca = PCA(n_components=2)
      pca.fit(X)
      X_transform = pca.transform(X)
      X_transform = pd.DataFrame(X_transform, columns=['PC1', 'PC2'])
      train_X, test_X, train_y, test_y = train_test_split(X_transform, y, test_size=0.
      ↪2, random_state=42)
```

4.3.1 Training set

```
[ ]: knn = KNN(k=247)
      knn.fit(train_X.values, train_y.values)
      pred_y = knn.predict(train_X.values)

[ ]: print(f'accuracy: {np.mean(train_y == pred_y)}')
```

accuracy: 0.6252379490328523

4.3.2 Test set

```
[ ]: pred_y = knn.predict(test_X.values)
      print(f'accuracy: {np.mean(test_y == pred_y)}')
```

accuracy: 0.6240379891927297

KNN perform worse than logistic regression and Decision Tree. I choose couple k values: 7,11,21,41 they perform similarly as k=247. The best accuracy i got from KNN is 0.71 which is still worse than other models.

4.3.3 important feature

KNN cannot find the most important features

5 Model selection

- Logistic Regression:

- run time: 1min
 - Train: 0.815
 - Test: 0.811
- Decision Tree:
 - run time: 3hours
 - Train: 0.866
 - Test: 0.864
- KNN:
 - run time: 1min
 - Train: 0.625
 - Test: 0.624

The model I will choose is Decision Tree, It has the best performance compare with other two models, but it takes long time to train the model. In addition, Decision Tree can find the important features easily. I think I can optimize the run time using some data structure and using vectorization by numpy.