You are exploring the wilderness of *Mushroomia*, a land populated by a plethora of diverse fauna and flora. In particular, *Mushroomia* is known for its unparalleled variety in mushrooms. However, not all the mushrooms in *Mushroomia* are edible. As you make your way through *Mushroomia*, you would like to know which mushrooms are edible, in order to forage for supplies for your daily mushroom soup.

You have access to:

- *Shroomster Pro Max* ^{TM} - a state of the art data collection device, developed by *Mushroomia*, that allows you to collect various data points about any mushroom you encounter in the wild
- *The National Archives on Mushrooms* - a dataset collected over the years by the government of *Mushroomia*

To address this problem, you decide to use the skills you learnt in CSM148 and train machine learning models on the *The National Archives on Mushrooms* in order to use your *Shroomster Pro Max* ^{TM} to determine whether the mushrooms you encounter on your adventure can be added to your daily mushroom soup.

This project will be more unstructured than the previous two projects in order to allow you to experience how data science problems are solved in practice. There are two parts to this project: a Jupyter Notebook with your code (where you explore, visualize, process your data and train machine learning models) and a report (where you explain the various choices you make in your implementation and analyze the final performance of your models).

1. Loading and Viewing Data

In [1]:
```
import numpy as np
import pandas as pd

df = pd.read_csv('mushroom_train.csv', sep=';')
df_test = pd.read_csv('mushroom_test.csv', sep=';')
```

In [2]:
```
df.head()
```

Out[2]:

	class	cap-diameter	cap-shape	cap-surface	cap-color	does-bruise-or-bleed	gill-attachment	gill-spacing	gill-color	stem-height	...	stem-root	stem-surface	stem-color	veil-type	veil-color	has-ring	ring-type
0	p	15.26	x	g	o	f	e	NaN	w	16.95	...	s	y	w	u	w	t	g
1	p	16.60	x	g	o	f	e	NaN	w	17.99	...	s	y	w	u	w	t	g
2	p	14.07	x	g	o	f	e	NaN	w	17.80	...	s	y	w	u	w	t	g
3	p	14.17	f	h	e	f	e	NaN	w	15.77	...	s	y	w	u	w	t	p
4	p	14.64	x	h	o	f	e	NaN	w	16.53	...	s	y	w	u	w	t	p

5 rows × 21 columns

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50213 entries, 0 to 50212
Data columns (total 21 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   class                 50213 non-null  object
 1   cap-diameter          50213 non-null  float64
 2   cap-shape             50213 non-null  object
 3   cap-surface           37915 non-null  object
 4   cap-color             50213 non-null  object
 5   does-bruise-or-bleed  50213 non-null  object
 6   gill-attachment       42447 non-null  object
 7   gill-spacing          31064 non-null  object
 8   gill-color            50213 non-null  object
 9   stem-height           50213 non-null  float64
 10  stem-width            50213 non-null  float64
 11  stem-root             7413 non-null   object
 12  stem-surface          19912 non-null  object
 13  stem-color            50213 non-null  object
 14  veil-type             3177 non-null   object
 15  veil-color            6297 non-null   object
 16  has-ring              50213 non-null  object
 17  ring-type             48448 non-null  object
 18  spore-print-color     4532 non-null   object
 19  habitat               50213 non-null  object
 20  season                50213 non-null  object
dtypes: float64(3), object(18)
memory usage: 8.0+ MB
```

In [4]:
```
df.isnull().sum()
```

Out[4]:

```
class                       0
cap-diameter                0
cap-shape                   0
cap-surface             12298
cap-color                   0
does-bruise-or-bleed        0
gill-attachment          7766
gill-spacing            19149
gill-color                  0
stem-height                 0
stem-width                  0
stem-root               42800
stem-surface            30301
stem-color                  0
veil-type               47036
veil-color              43916
has-ring                    0
ring-type                1765
spore-print-color       45681
habitat                     0
season                      0
dtype: int64
```

In [5]:

```
df_test.isnull().sum()
```

Out[5]:
```
class                       0
cap-diameter                0
cap-shape                   0
cap-surface              1822
cap-color                   0
does-bruise-or-bleed        0
gill-attachment          2118
gill-spacing             5914
gill-color                  0
stem-height                 0
stem-width                  0
stem-root                8738
stem-surface             7823
stem-color                  0
veil-type               10856
veil-color               9740
has-ring                    0
ring-type                 706
spore-print-color        9034
habitat                     0
season                      0
dtype: int64
```

It appears that we have null values in nine columns (all categorical). Almost all the columns with null values are missing a significant percentage (more than 20%) of their values, which is potentially concerning. Five of the columns are missing more than half of their values, with four of them missing more than 80%. These features should raise some serious concern.

In [6]:
```
df.describe()
```

Out[6]:

	cap-diameter	stem-height	stem-width
count	50213.000000	50213.000000	50213.000000
mean	6.245186	6.600561	10.763191
std	4.542552	3.221714	7.744992
min	0.380000	1.200000	0.520000
25%	3.290000	4.680000	4.720000
50%	5.540000	5.900000	9.130000
75%	8.100000	7.600000	15.210000
max	58.890000	33.920000	58.950000

In [7]:
```
df['class'].unique()
```

Out[7]:
```
array(['p', 'e'], dtype=object)
```

2. Splitting Data into Features and Labels

In [8]:
```
X_train = df.drop('class', axis=1).copy()
X_test = df_test.drop('class', axis=1).copy()
X_train.head()
```

Out[8]:

	cap-diameter	cap-shape	cap-surface	cap-color	does-bruise-or-bleed	gill-attachment	gill-spacing	gill-color	stem-height	stem-width	stem-root	stem-surface	stem-color	veil-type	veil-color	has-ring	ring-type
0	15.26	x	g	o	f	e	NaN	w	16.95	17.09	s	y	w	u	w	t	g
1	16.60	x	g	o	f	e	NaN	w	17.99	18.19	s	y	w	u	w	t	g
2	14.07	x	g	o	f	e	NaN	w	17.80	17.74	s	y	w	u	w	t	g
3	14.17	f	h	e	f	e	NaN	w	15.77	15.98	s	y	w	u	w	t	p
4	14.64	x	h	o	f	e	NaN	w	16.53	17.20	s	y	w	u	w	t	p

In [9]:
```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50213 entries, 0 to 50212
Data columns (total 20 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   cap-diameter          50213 non-null  float64
 1   cap-shape             50213 non-null  object
 2   cap-surface           37915 non-null  object
 3   cap-color             50213 non-null  object
 4   does-bruise-or-bleed  50213 non-null  object
 5   gill-attachment       42447 non-null  object
 6   gill-spacing          31064 non-null  object
 7   gill-color            50213 non-null  object
 8   stem-height           50213 non-null  float64
 9   stem-width            50213 non-null  float64
 10  stem-root             7413 non-null   object
 11  stem-surface          19912 non-null  object
 12  stem-color            50213 non-null  object
 13  veil-type             3177 non-null   object
 14  veil-color            6297 non-null   object
 15  has-ring              50213 non-null  object
 16  ring-type             48448 non-null  object
 17  spore-print-color     4532 non-null   object
 18  habitat               50213 non-null  object
 19  season                50213 non-null  object
dtypes: float64(3), object(17)
memory usage: 7.7+ MB
```

At this time, we'll also convert the labels to binary labels using p=0, e=1 as specified by the project spec. This will make the data visualization in the next part easier to manage.

In [10]:
```
# Convert labels to binary labels using p=0, e=1 as specified by the project spec
y_train = np.zeros_like(df['class'], dtype=np.float64)
y_train[df['class'] == 'e'] = 1
y_test = np.zeros_like(df_test['class'], dtype=np.float64)
y_test[df_test['class'] == 'e'] = 1
```

In [11]:
```
y_avg = np.mean(y_train)
y_avg
```

Out[11]:
```
0.41764084997908907
```

3. Data Exploration and Visualization

In [12]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

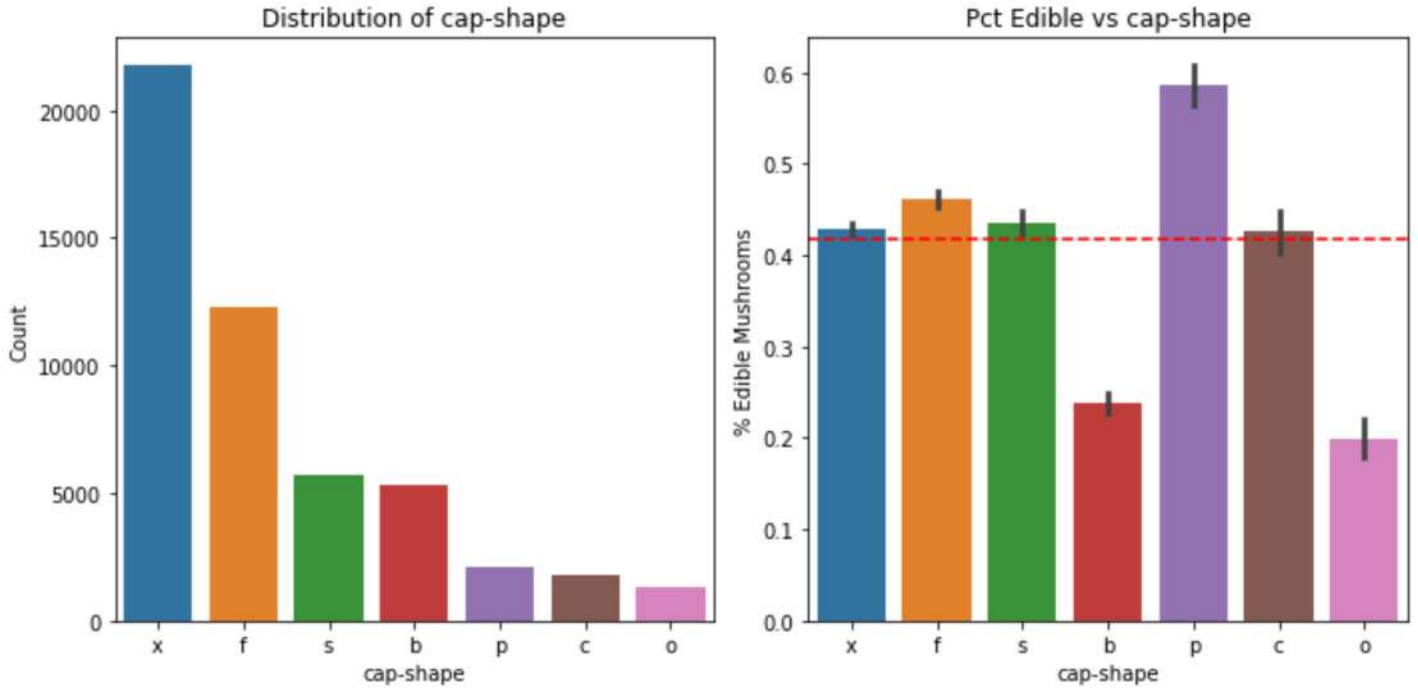
To make visualization easier, let's start by replacing NaN values with a placeholder character. We'll use '?' since it doesn't appear in any of the categorical variables. Thankfully, none of the numerical features are missing values.

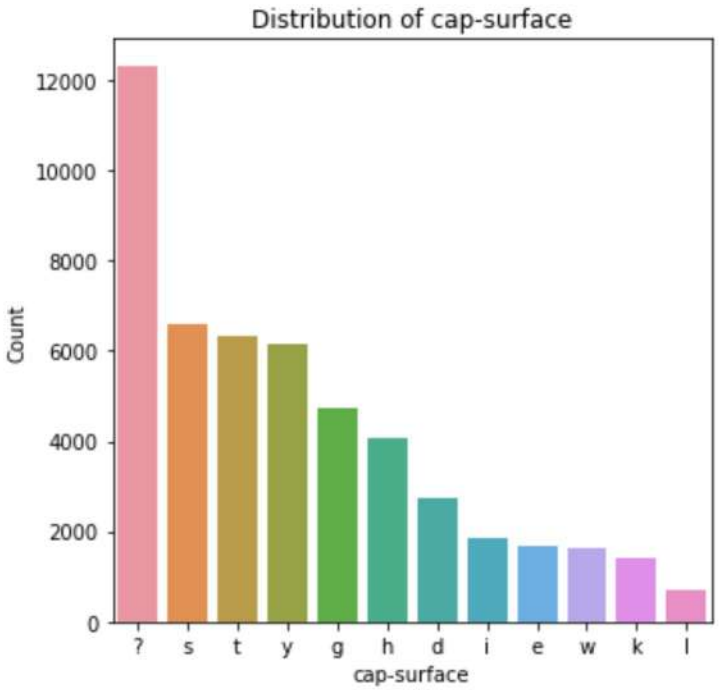
In [13]:
```
for label in X_train.columns[X_train.isnull().sum() != 0]:
    X_train.loc[X_train[label].isnull(), label] = '?'
    X_test.loc[X_test[label].isnull(), label] = '?' # Do the same to X_test so pipelining works properly
X_train.isnull().sum()
```

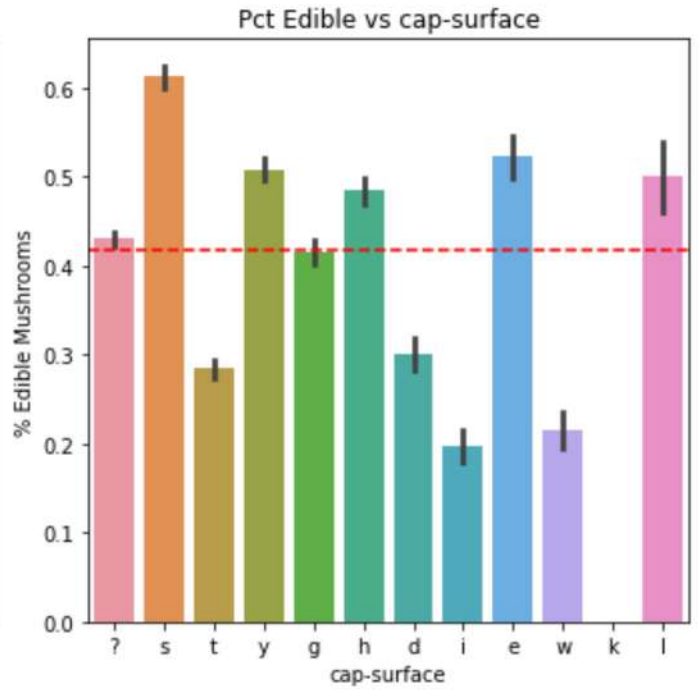
Out[13]:
```
cap-diameter            0
cap-shape               0
cap-surface             0
cap-color               0
does-bruise-or-bleed    0
gill-attachment         0
gill-spacing            0
gill-color              0
stem-height             0
stem-width              0
stem-root               0
stem-surface            0
stem-color              0
veil-type               0
veil-color              0
has-ring                0
ring-type               0
spore-print-color       0
habitat                 0
season                  0
dtype: int64
```

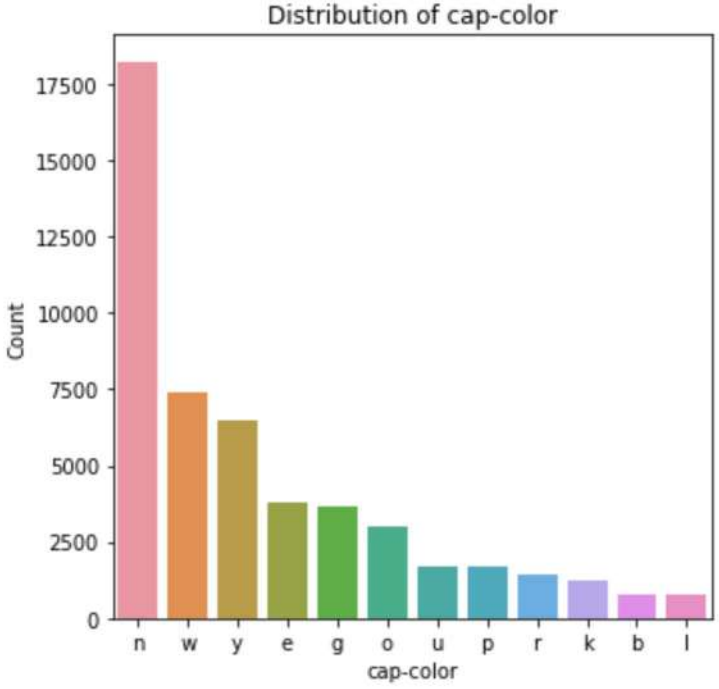
Let's explore the relative frequency of edible mushrooms in each of the categorical classes.

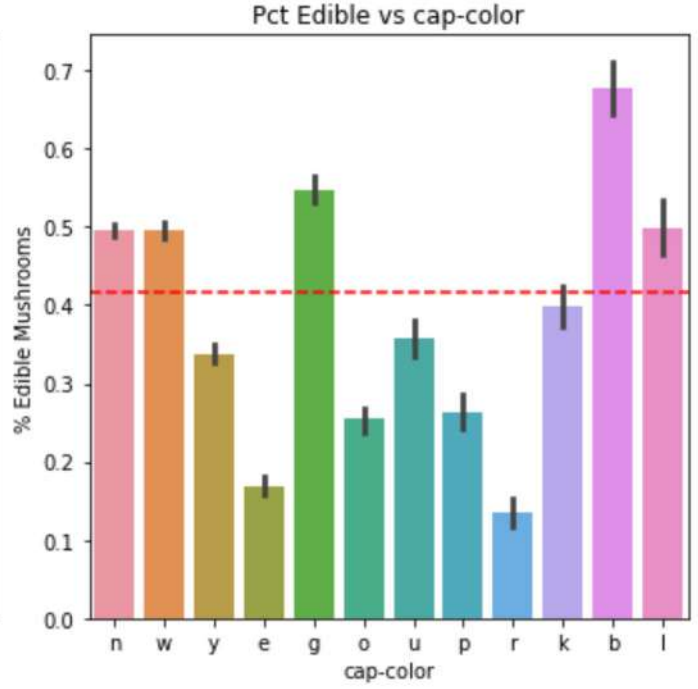
In [14]:
```
categorical_feat = X_train.dtypes[X_train.dtypes == 'object'].index.values
for label in categorical_feat:
    fig, ax = plt.subplots(1, 2, figsize=(10,5))
    # Order by counts for easier visualization
    order = X_train[label].value_counts().sort_values(ascending=False).index
    sns.countplot(data=X_train, x=label, order=order, ax=ax[0])
    ax[0].set_ylabel('Count')
    ax[0].set_title('Distribution of ' + label)
    sns.barplot(data=X_train, x=label, y=y_train, order=order, ax=ax[1])
    # Mark a line at the average percent for the entire training dataset
    ax[1].axhline(y=y_avg, color='red', linestyle='dashed')
    ax[1].set_ylabel('% Edible Mushrooms')
    ax[1].set_title('Pct Edible vs ' + label)
    fig.tight_layout()
    plt.show()
```

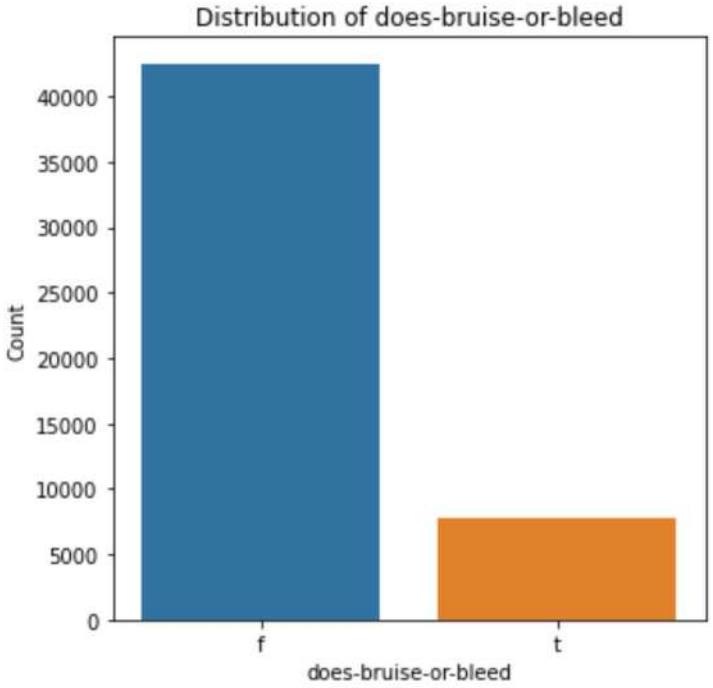


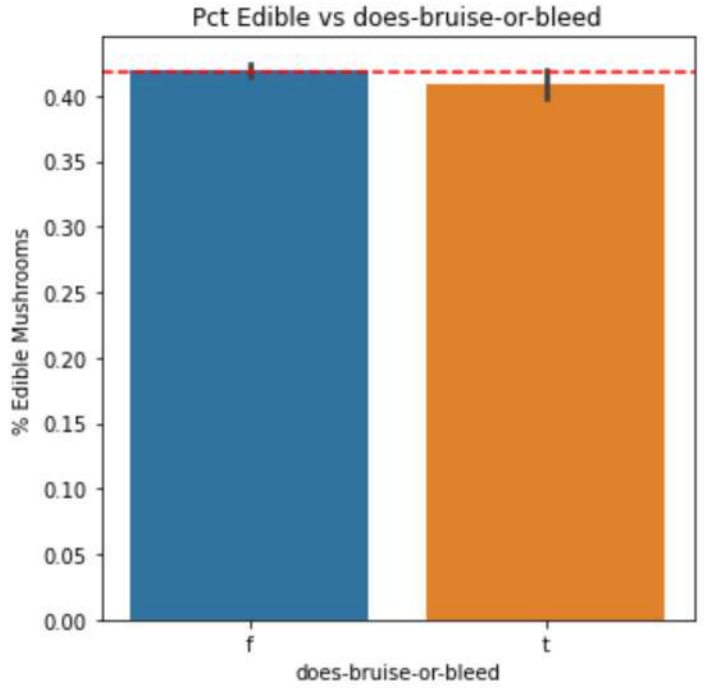

Distribution of cap-surface
Count
cap-surface
? s t y g h d i e w k l
0 2000 4000 6000 8000 10000 12000

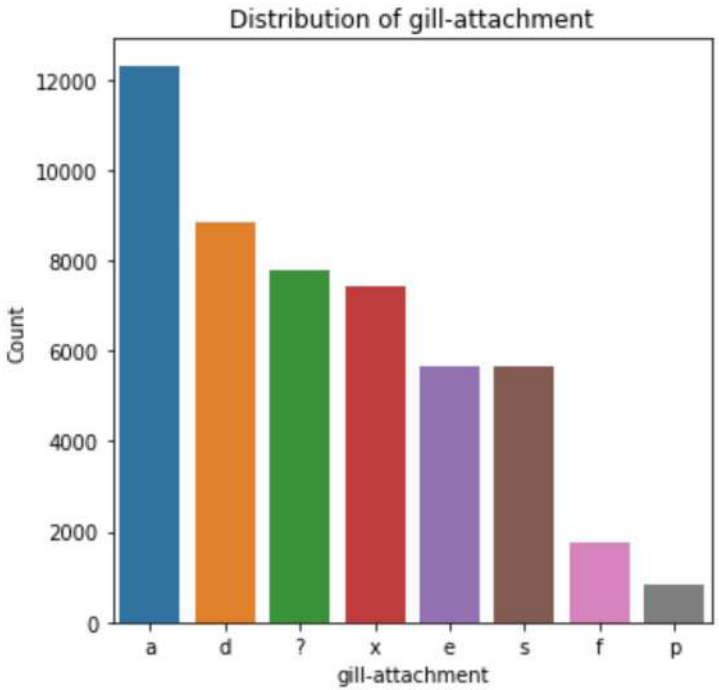

Pct Edible vs cap-surface
% Edible Mushrooms
cap-surface
? s t y g h d i e w k l
0.0 0.1 0.2 0.3 0.4 0.5 0.6

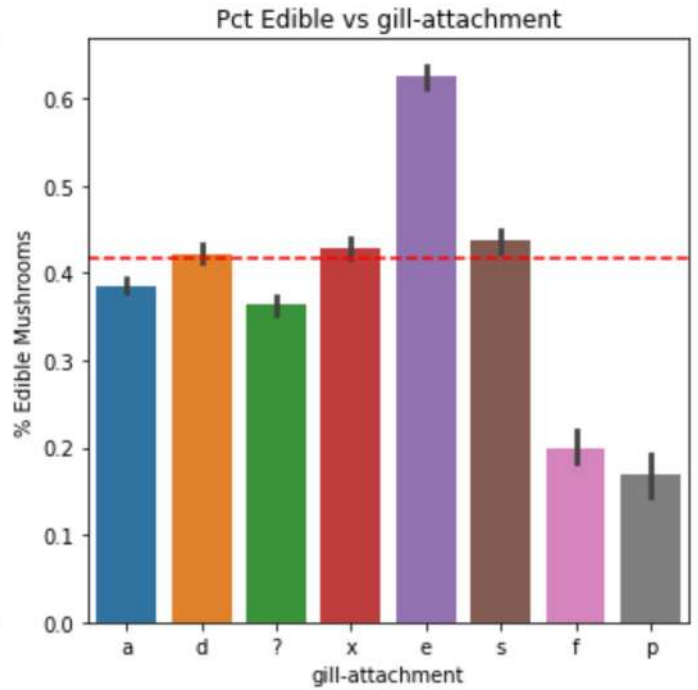

Distribution of cap-color
Count
cap-color
n w y e g o u p r k b l
0 2500 5000 7500 10000 12500 15000 17500

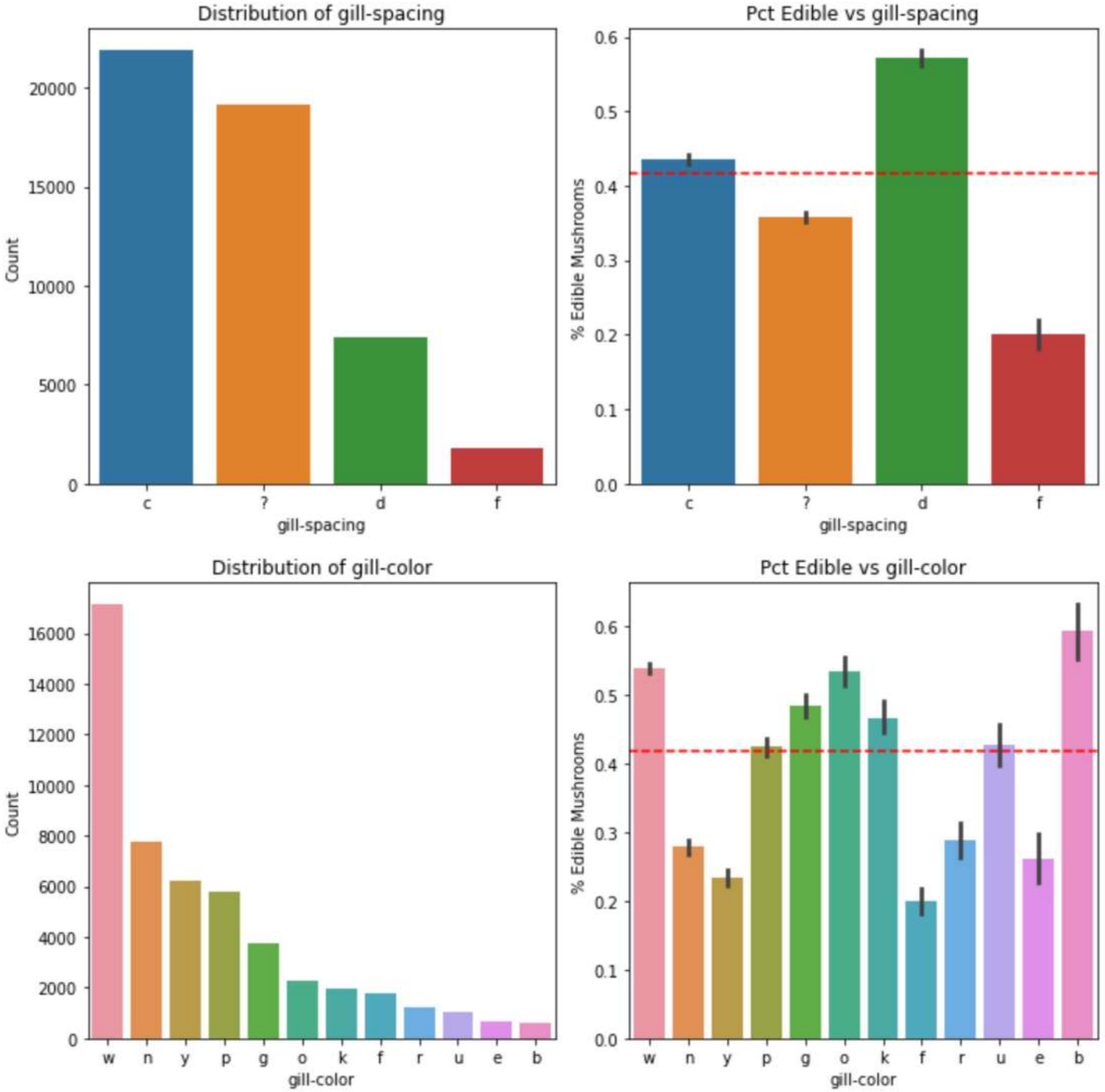

Pct Edible vs cap-color
% Edible Mushrooms
cap-color
n w y e g o u p r k b l
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7

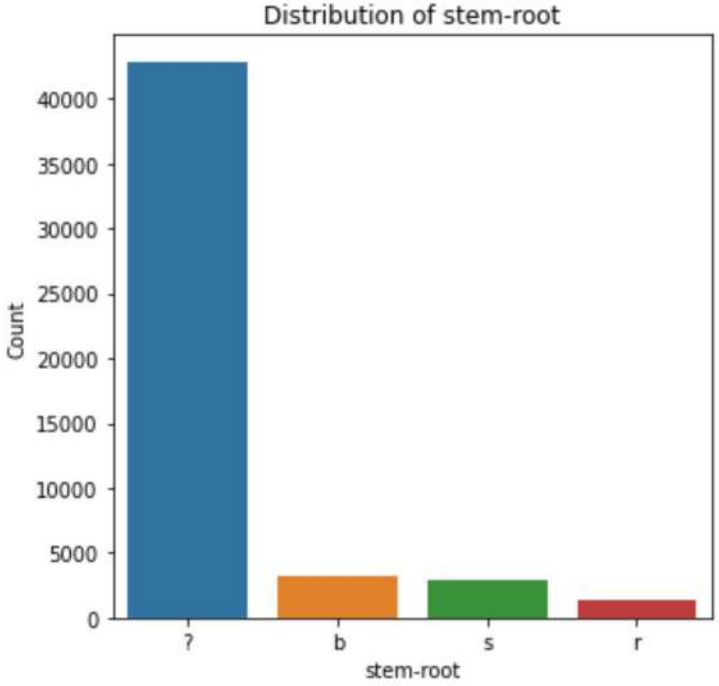

Distribution of does-bruise-or-bleed
Count
40000
35000
30000
25000
20000
15000
10000
5000
0
f
t
does-bruise-or-bleed

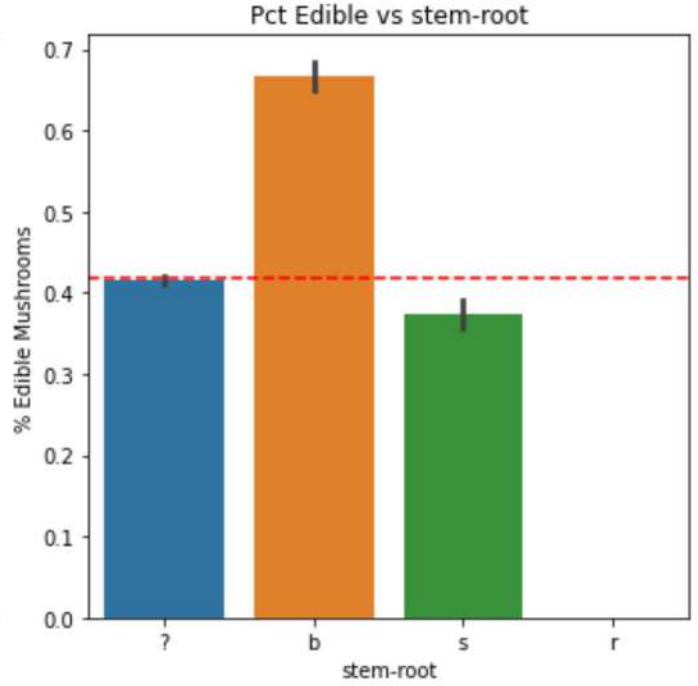

Pct Edible vs does-bruise-or-bleed
% Edible Mushrooms
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
f
t
does-bruise-or-bleed

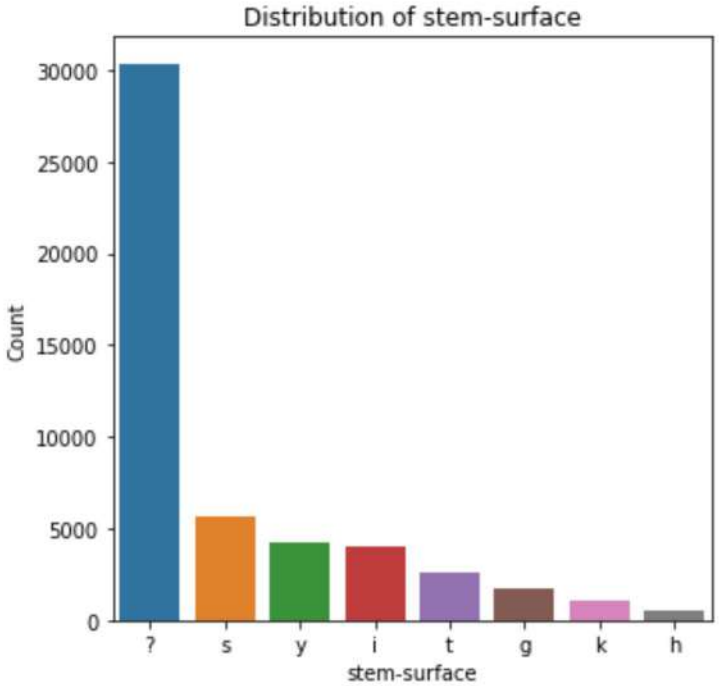

Distribution of gill-attachment
Count
12000
10000
8000
6000
4000
2000
0
a
d
?
x
e
s
f
p
gill-attachment

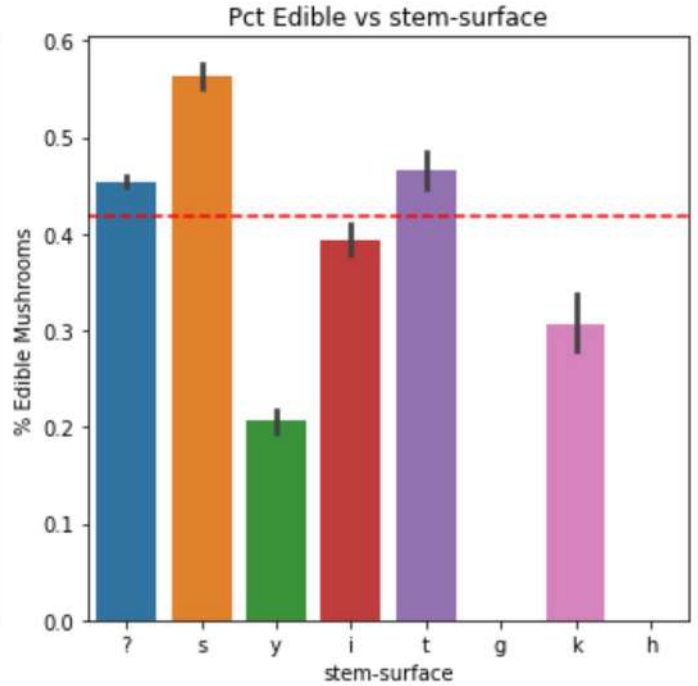

Pct Edible vs gill-attachment
% Edible Mushrooms
0.6
0.5
0.4
0.3
0.2
0.1
0.0
a
d
?
x
e
s
f
p
gill-attachment

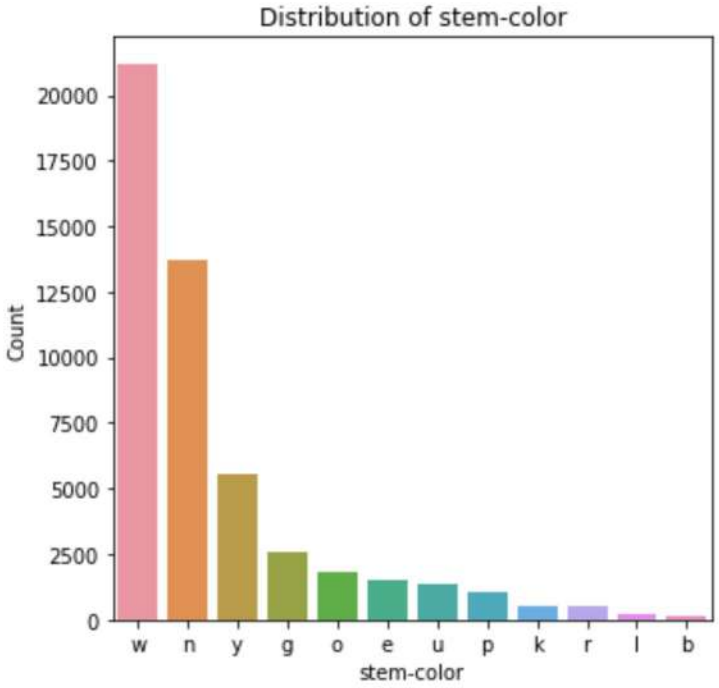

Distribution of gill-spacing
Count
gill-spacing
c
?
d
f
Pct Edible vs gill-spacing
% Edible Mushrooms
gill-spacing
Distribution of gill-color
Count
gill-color
w
n
y
p
g
o
k
f
r
u
e
b
Pct Edible vs gill-color
% Edible Mushrooms
gill-color

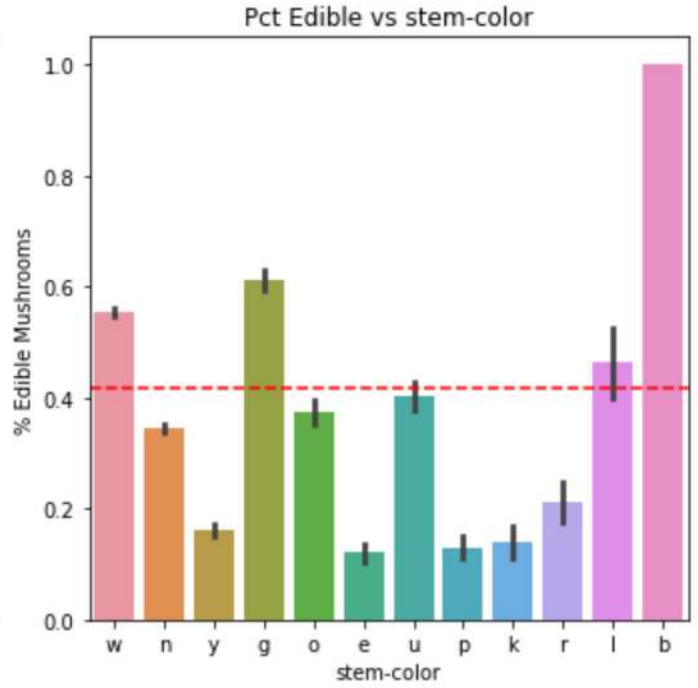

Distribution of stem-root
Count
stem-root
?
b
s
r

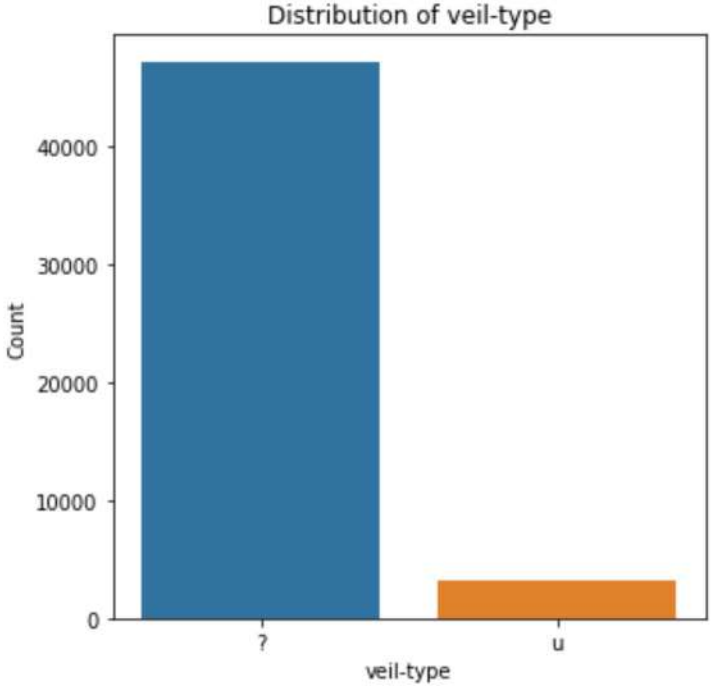

Pct Edible vs stem-root
% Edible Mushrooms
stem-root
?
b
s
r

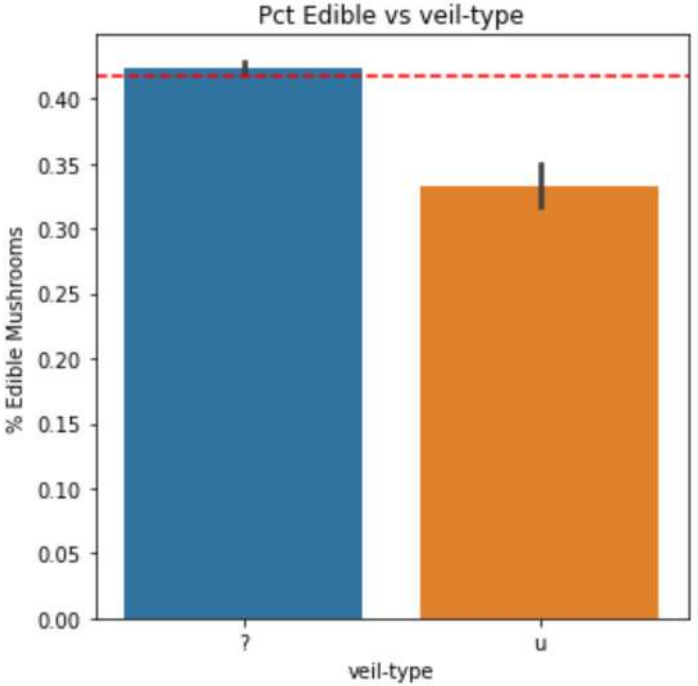

Distribution of stem-surface
Count
stem-surface
?
s
y
i
t
g
k
h

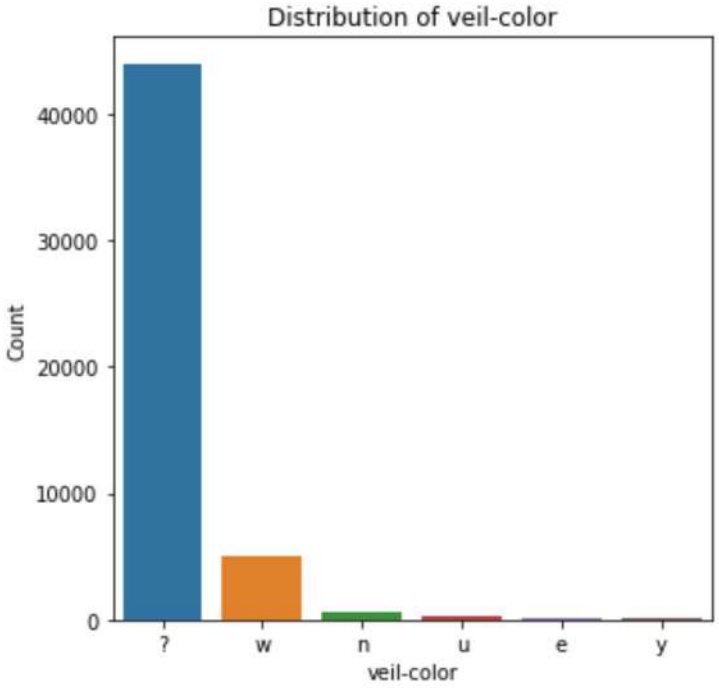

Pct Edible vs stem-surface
% Edible Mushrooms
stem-surface
?
s
y
i
t
g
k
h

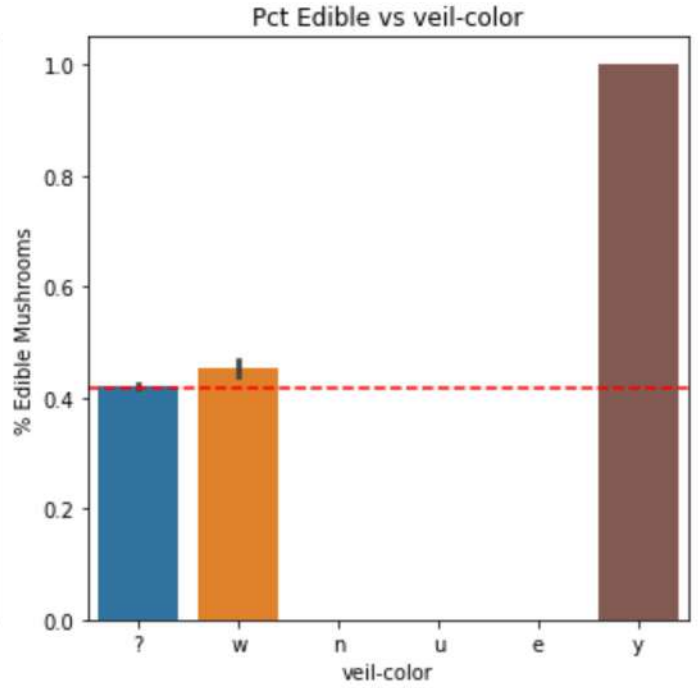

Distribution of stem-color
Count
stem-color
w n y g o e u p k r l b
0 2500 5000 7500 10000 12500 15000 17500 20000

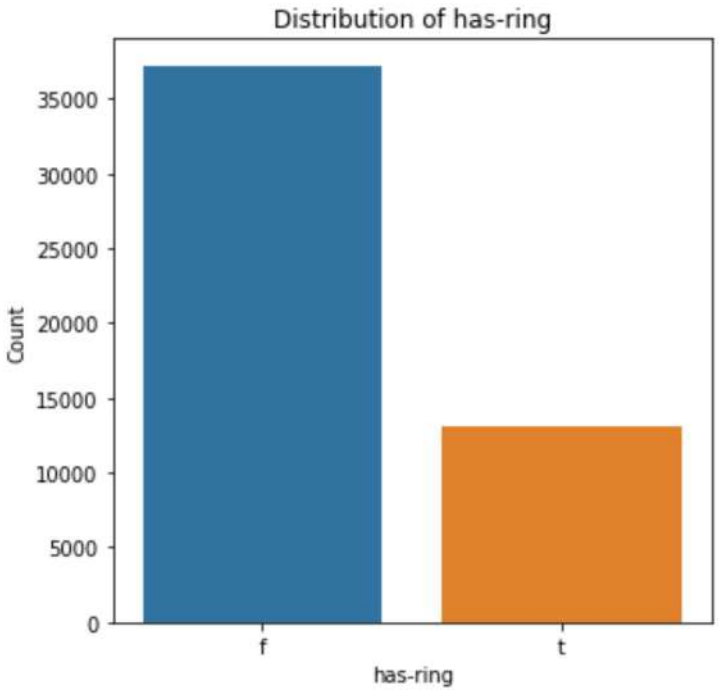

Pct Edible vs stem-color
% Edible Mushrooms
stem-color
w n y g o e u p k r l b
0.0 0.2 0.4 0.6 0.8 1.0

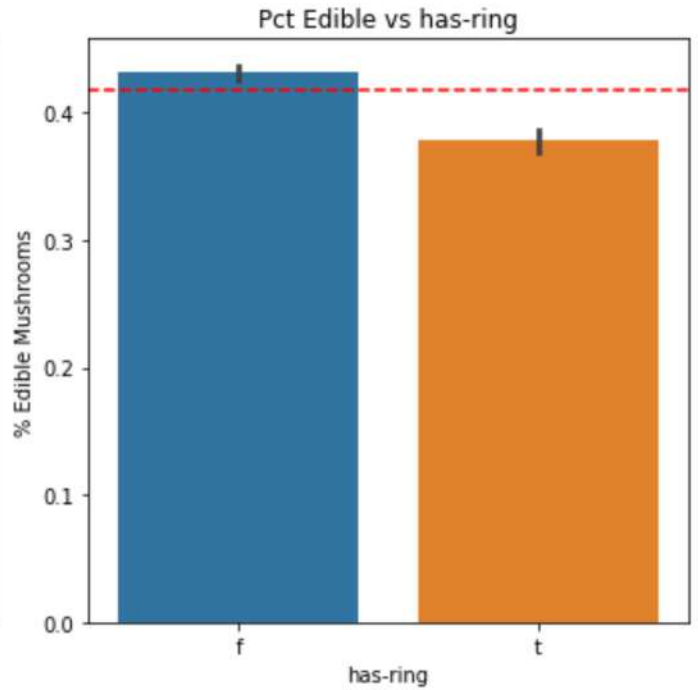

Distribution of veil-type
Count
veil-type
? u
0 10000 20000 30000 40000

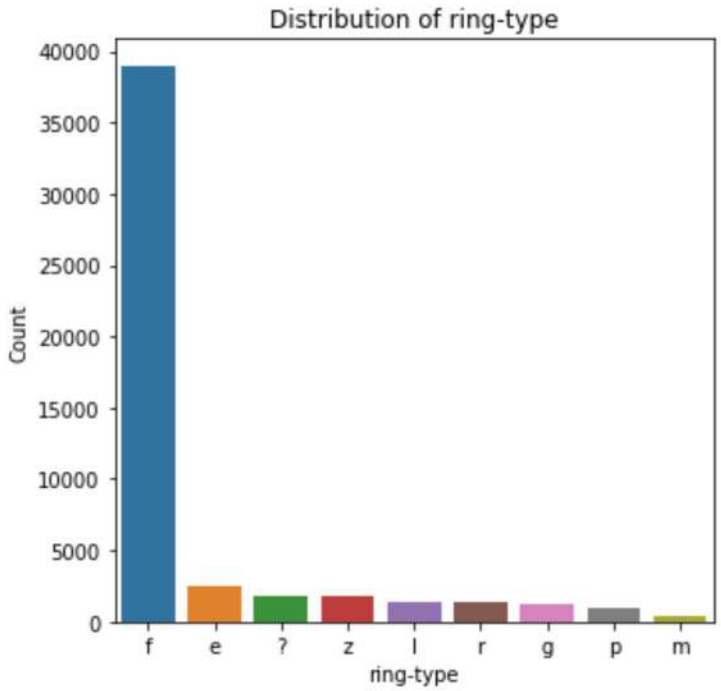

Pct Edible vs veil-type
% Edible Mushrooms
veil-type
? u
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40

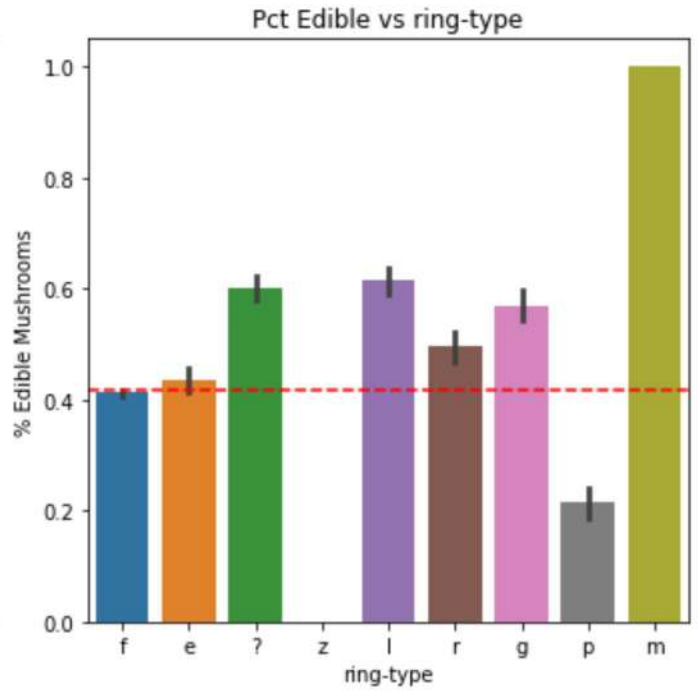

Distribution of veil-color
Count
40000
30000
20000
10000
0
?
w
n
u
e
y
veil-color

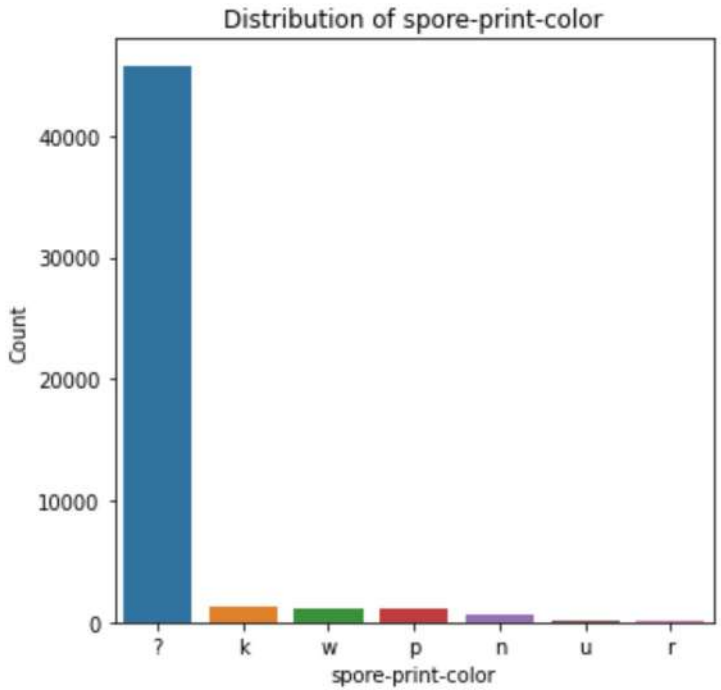

Pct Edible vs veil-color
% Edible Mushrooms
1.0
0.8
0.6
0.4
0.2
0.0
?
w
n
u
e
y
veil-color

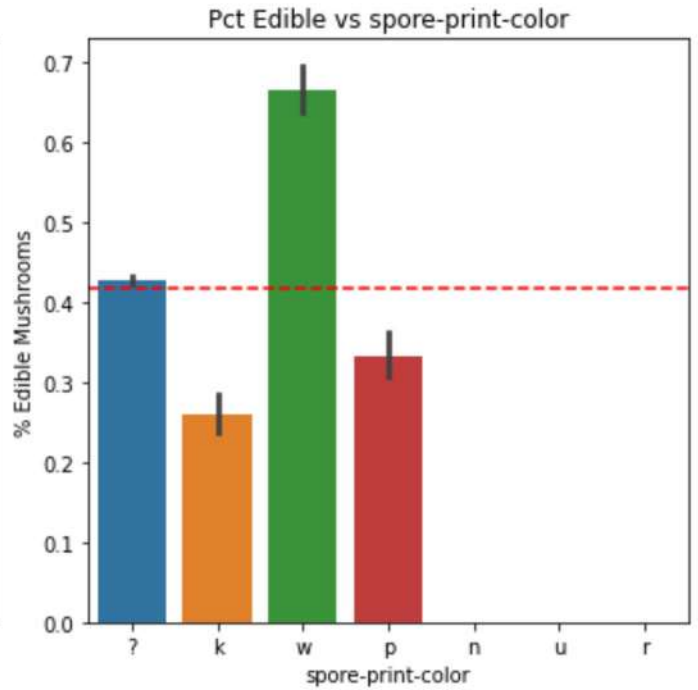

Distribution of has-ring
Count
35000
30000
25000
20000
15000
10000
5000
0
f
t
has-ring

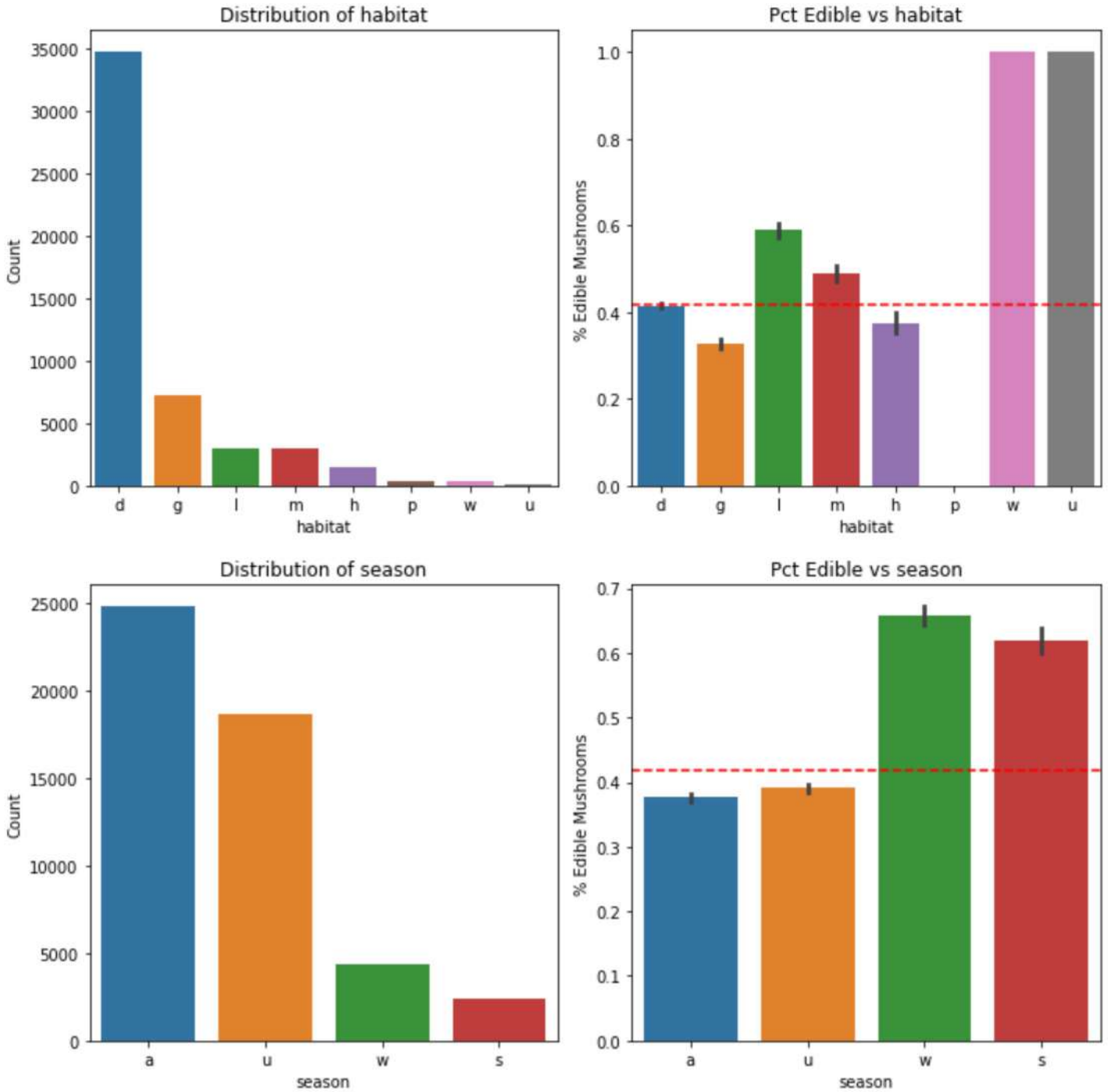

Pct Edible vs has-ring
% Edible Mushrooms
0.4
0.3
0.2
0.1
0.0
f
t
has-ring


Distribution of ring-type
Count
ring-type
f
e
?
z
l
r
g
p
m


Pct Edible vs ring-type
% Edible Mushrooms
ring-type


Distribution of spore-print-color
Count
spore-print-color
?
k
w
p
n
u
r

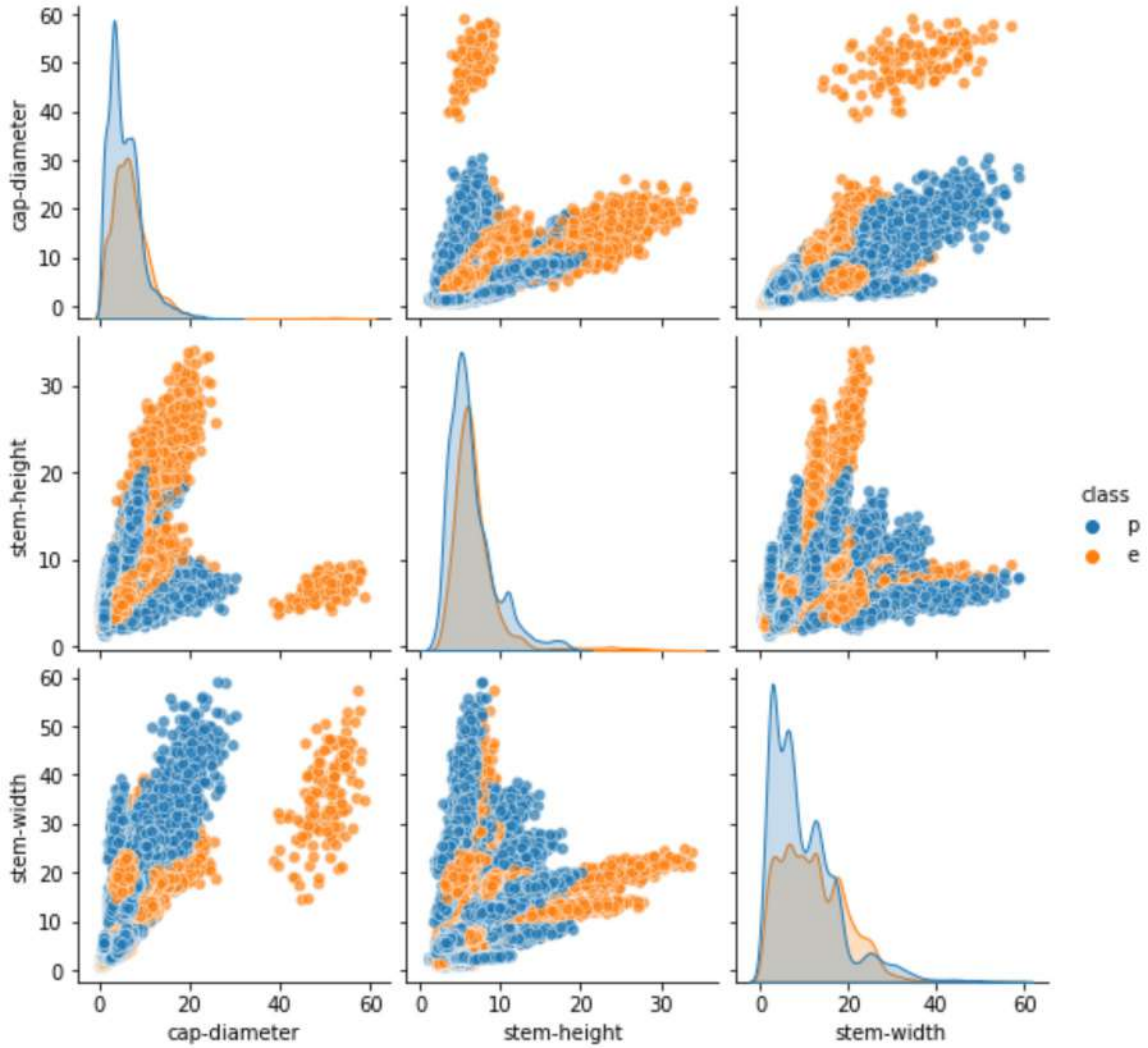

Pct Edible vs spore-print-color
% Edible Mushrooms
spore-print-color


Distribution of habitat
Count
habitat
d
g
l
m
h
p
w
u
Pct Edible vs habitat
% Edible Mushrooms
habitat
Distribution of season
Count
season
a
u
w
s
Pct Edible vs season
% Edible Mushrooms
season

From the categorical distributions, we can see that most categories have a percentage of edible mushrooms that falls relatively close to the mean. This makes sense for variables where one category nearly completely dominates the entire dataset, like `habitat`. A few variables have categories where the entire category is the same class, either 0 or 1. There appear to be about 17 such categories across the 17 categorical variables, mostly extremely infrequent categories, like categories 'p', 'w', and 'u' in `habitat`.

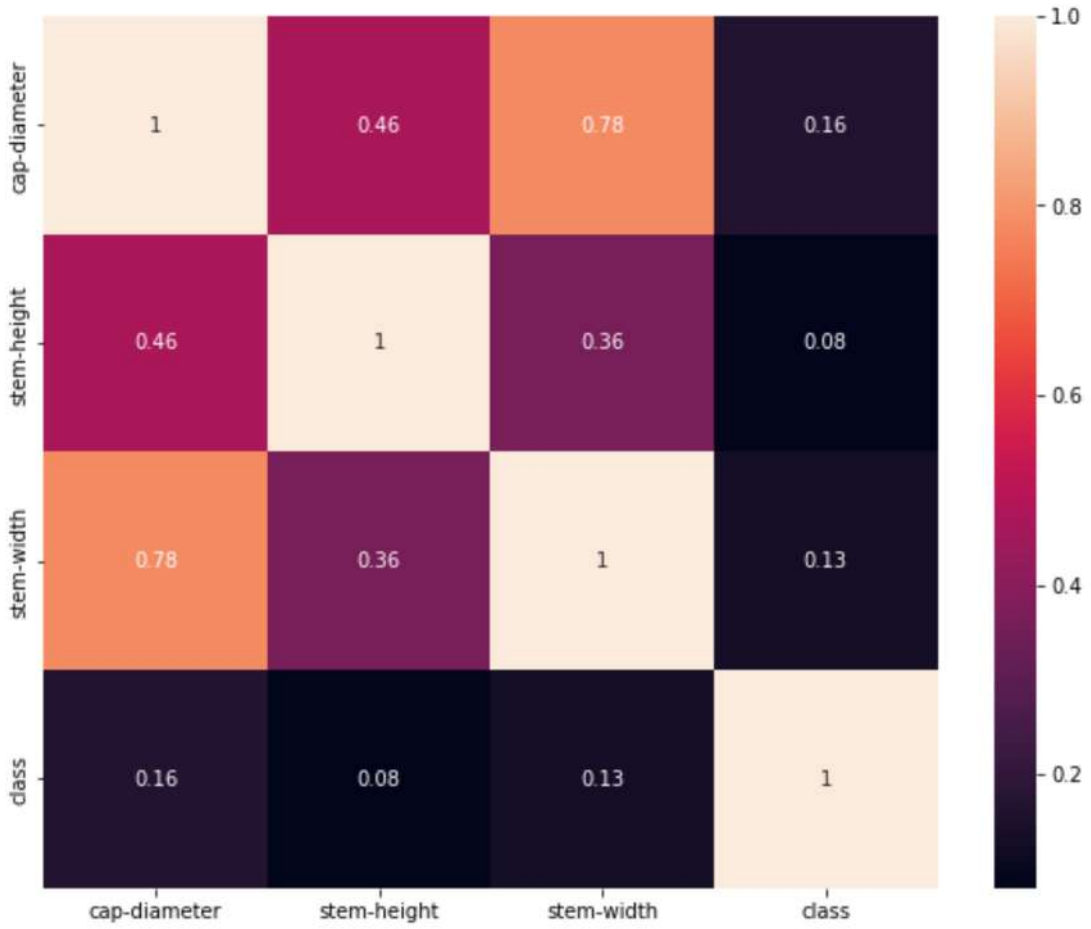
Now we turn to the (thankfully fewer) numerical categories.

In [15]:
```
numerical_feat = X_train.dtypes[X_train.dtypes != 'object'].index.values
sns.pairplot(data=df[np.append(numerical_feat, 'class')], hue='class', plot_kws={'alpha': 0.7});
```



From the pairwise plots, we can see that the distributions of the numerical variables overlap quite heavily for both classes. However, the pairwise plots reveal some small but very distinct groupings that clearly separate portions of the edible mushrooms from the rest of the data. This appears to be the case for very high values of `cap-diameter` and `stem-height` .

In [16]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 8))
df_corr = df[numerical_feat].copy()
df_corr['class'] = y_train
sns.heatmap(df_corr.corr(), ax=ax, annot=True);
```



From the correlation plot, we can see high correlations between `stem-width` and `cap-diameter`, and moderate correlations between the other two pairs. The correlations with `class` are very low, but slightly positive. This is to be expected from the overlap in the densities estimated in the plots above. The slight positive effect is likely due to the longer right tails that we can see for the features in the density plots.

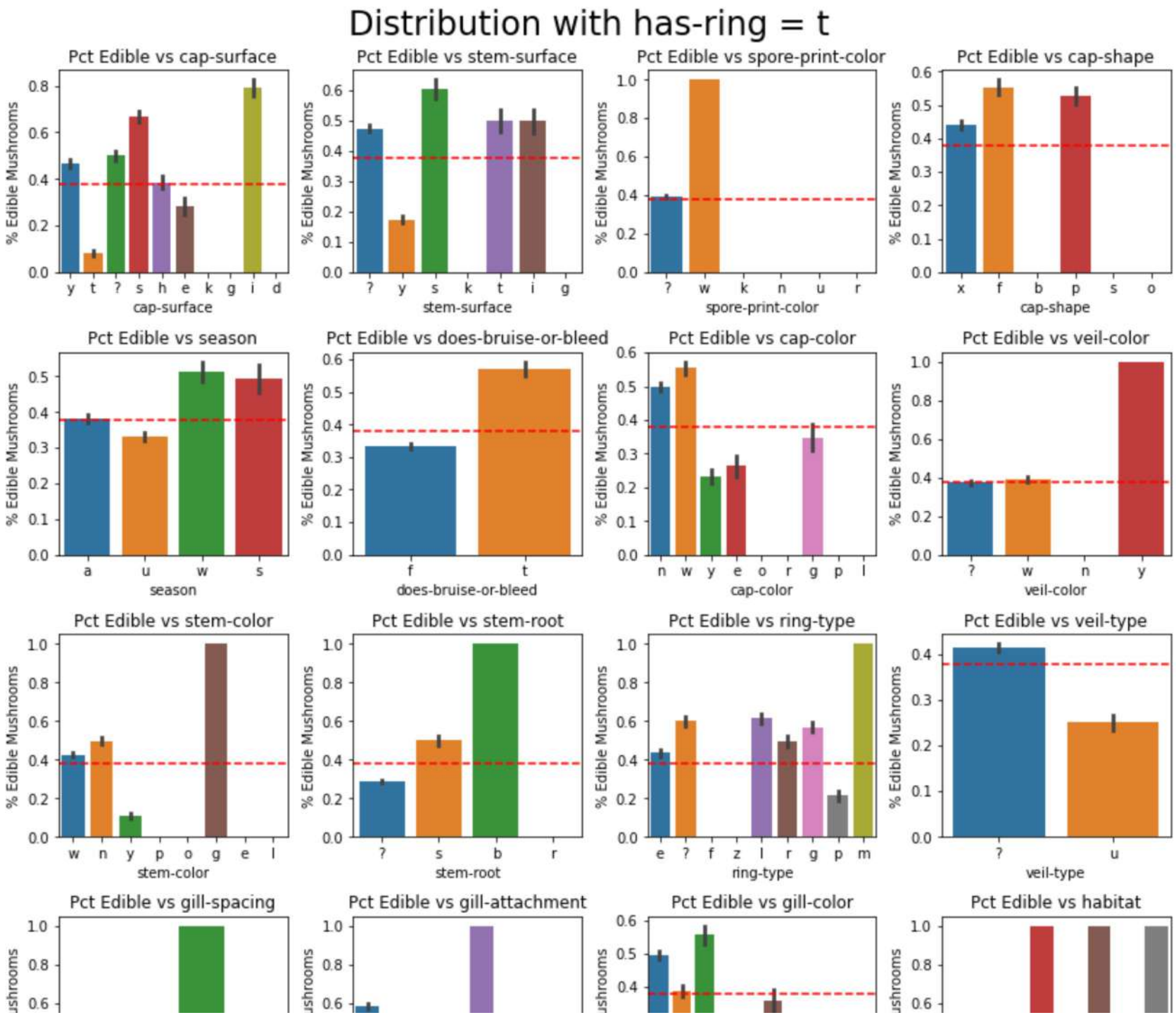
5. Data Augmentation (Creating at least 2 New Features)

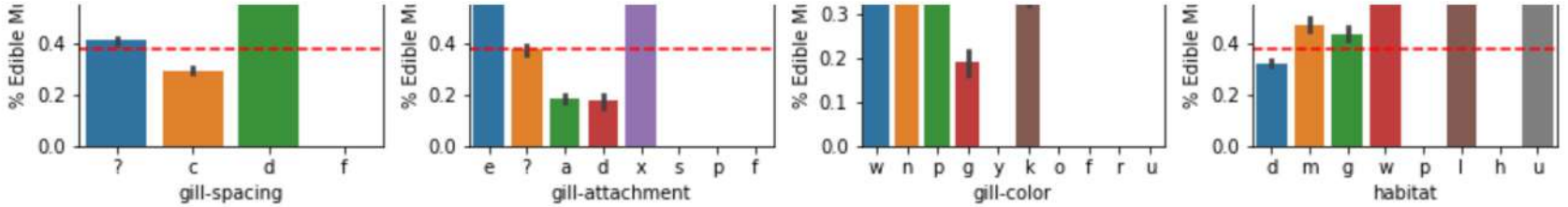
Let's explore how the distribution of class labels changes when we examine a subset of the data corresponding to one of the binary features:

In [17]:

```
X_exploratory = X_train[X_train['has-ring'] == 't']
y_exploratory = y_train[X_train['has-ring'] == 't']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['has-ring'])):
    if label != 'has-ring':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with has-ring = t", fontsize=24)
fig.tight_layout()
plt.show()
```

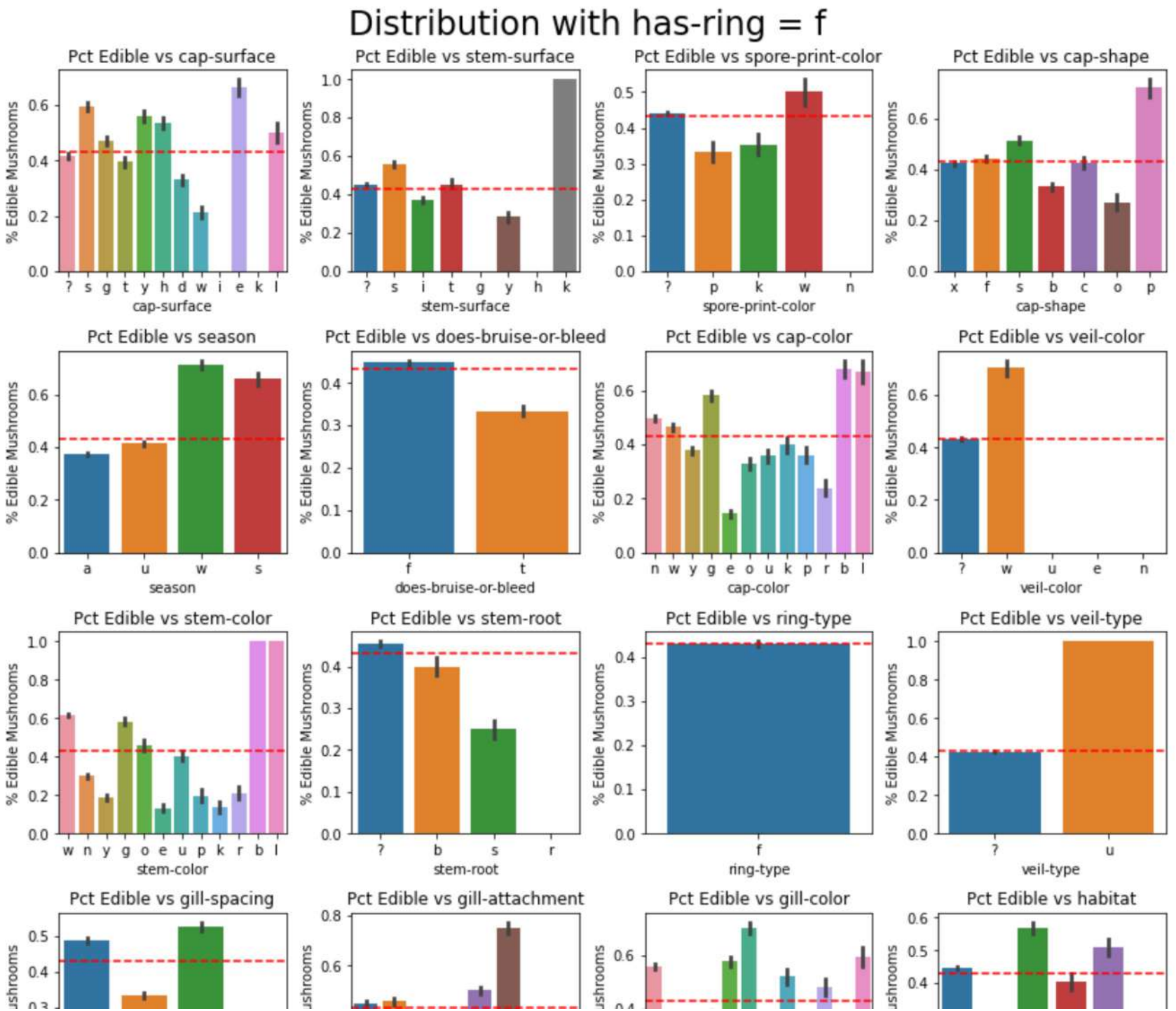

Distribution with has-ring = t
Pct Edible vs cap-surface
% Edible Mushrooms
y t ? s h e k g i d
cap-surface
Pct Edible vs stem-surface
? y s k t i g
stem-surface
Pct Edible vs spore-print-color
? w k n u r
spore-print-color
Pct Edible vs cap-shape
x f b p s o
cap-shape
Pct Edible vs season
a u w s
season
Pct Edible vs does-bruise-or-bleed
f t
does-bruise-or-bleed
Pct Edible vs cap-color
n w y e o r g p l
cap-color
Pct Edible vs veil-color
? w n y
veil-color
Pct Edible vs stem-color
w n y p o g e l
stem-color
Pct Edible vs stem-root
? s b r
stem-root
Pct Edible vs ring-type
e ? f z l r g p m
ring-type
Pct Edible vs veil-type
? u
veil-type
Pct Edible vs gill-spacing
Pct Edible vs gill-attachment
Pct Edible vs gill-color
Pct Edible vs habitat

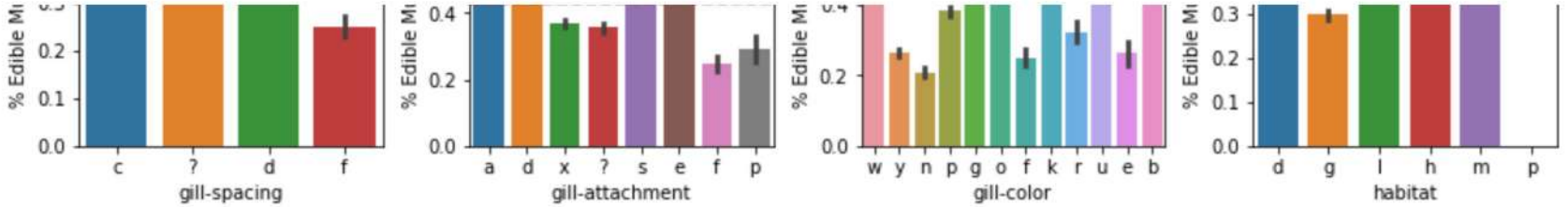


In [18]:

```
X_exploratory = X_train[X_train['has-ring'] == 'f']
y_exploratory = y_train[X_train['has-ring'] == 'f']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['has-ring'])):
    if label != 'has-ring':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with has-ring = f", fontsize=24)
fig.tight_layout()
plt.show()
```


Distribution with has-ring = f
Pct Edible vs cap-surface
% Edible Mushrooms
? s g t y h d w i e k l
cap-surface
Pct Edible vs stem-surface
% Edible Mushrooms
? s i t g y h k
stem-surface
Pct Edible vs spore-print-color
% Edible Mushrooms
? p k w n
spore-print-color
Pct Edible vs cap-shape
% Edible Mushrooms
x f s b c o p
cap-shape
Pct Edible vs season
% Edible Mushrooms
a u w s
season
Pct Edible vs does-bruise-or-bleed
% Edible Mushrooms
f t
does-bruise-or-bleed
Pct Edible vs cap-color
% Edible Mushrooms
n w y g e o u k p r b l
cap-color
Pct Edible vs veil-color
% Edible Mushrooms
? w u e n
veil-color
Pct Edible vs stem-color
% Edible Mushrooms
w n y g o e u p k r b l
stem-color
Pct Edible vs stem-root
% Edible Mushrooms
? b s r
stem-root
Pct Edible vs ring-type
% Edible Mushrooms
f
ring-type
Pct Edible vs veil-type
% Edible Mushrooms
? u
veil-type
Pct Edible vs gill-spacing
Pct Edible vs gill-attachment
Pct Edible vs gill-color
Pct Edible vs habitat

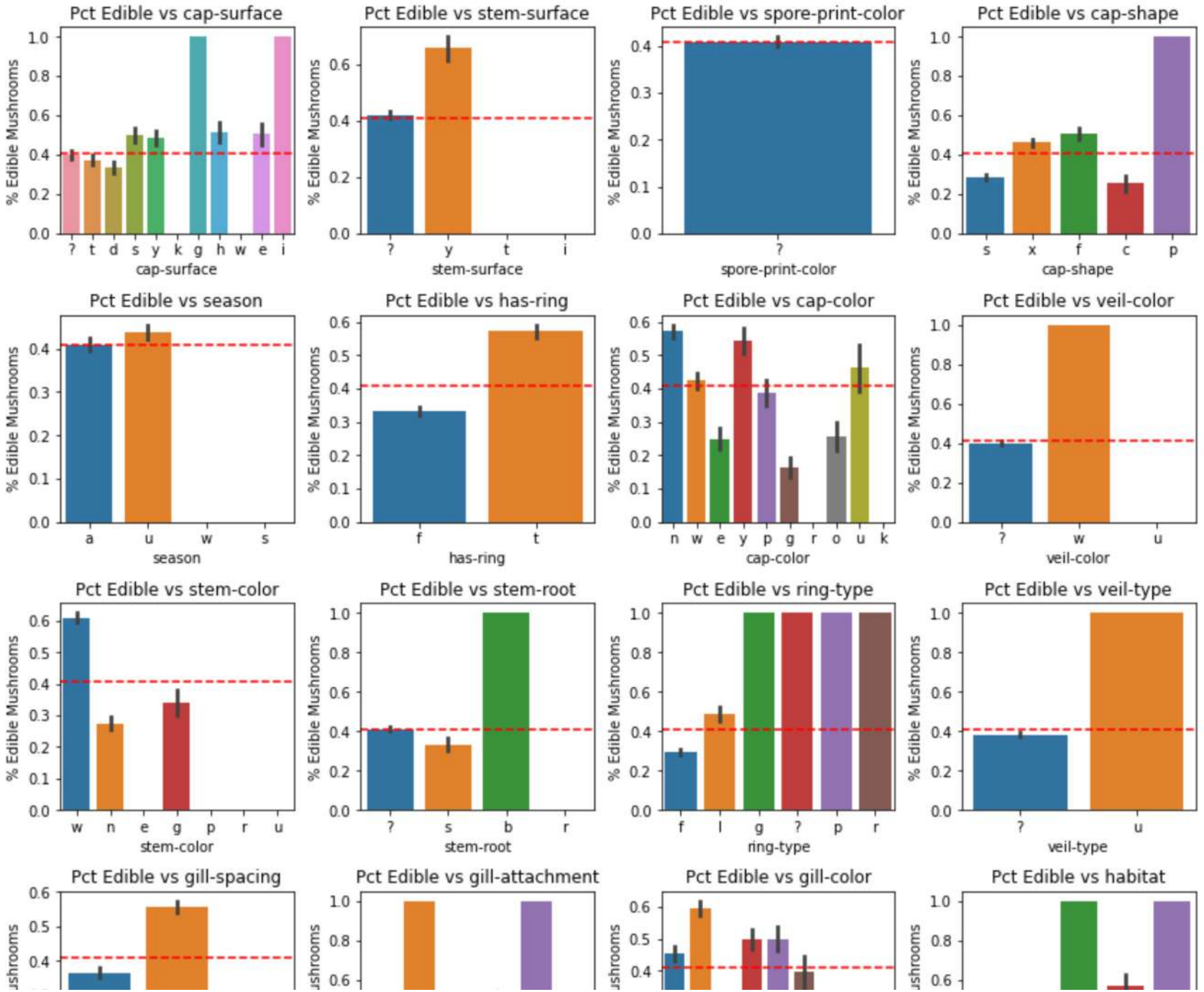


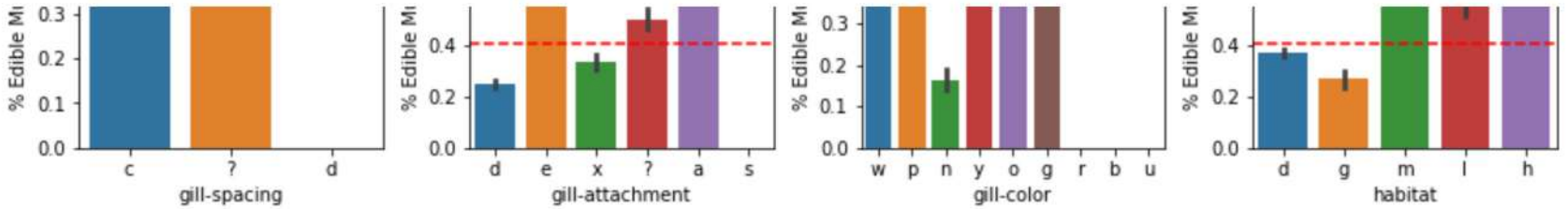
In [19]:

```
X_exploratory = X_train[X_train['does-bruise-or-bleed'] == 't']
y_exploratory = y_train[X_train['does-bruise-or-bleed'] == 't']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['does-bruise-or-bleed'])):
    if label != 'does-bruise-or-bleed':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with does-bruise-or-bleed = t", fontsize=24)
fig.tight_layout()
plt.show()
```

Distribution with does-bruise-or-bleed = t



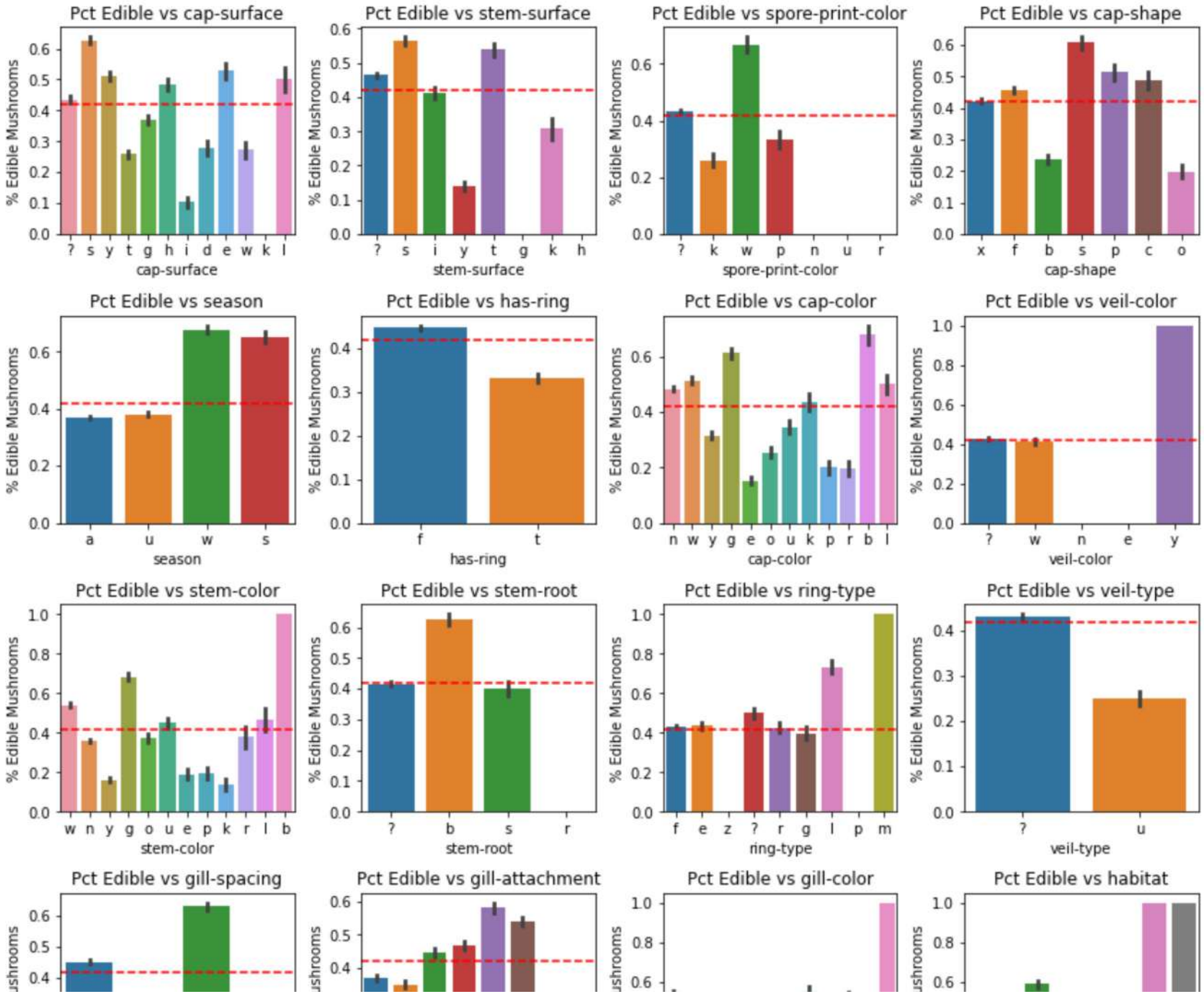


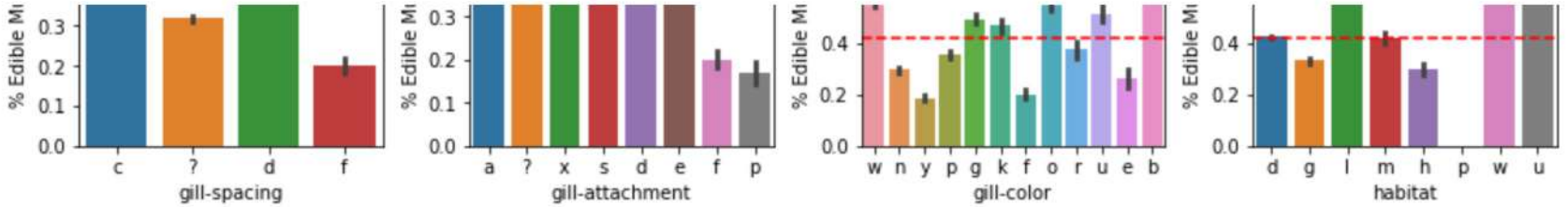
In [20]:

```
X_exploratory = X_train[X_train['does-bruise-or-bleed'] == 'f']
y_exploratory = y_train[X_train['does-bruise-or-bleed'] == 'f']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['does-bruise-or-bleed'])):
    if label != 'does-bruise-or-bleed':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with does-bruise-or-bleed = f", fontsize=24)
fig.tight_layout()
plt.show()
```

Distribution with does-bruise-or-bleed = f

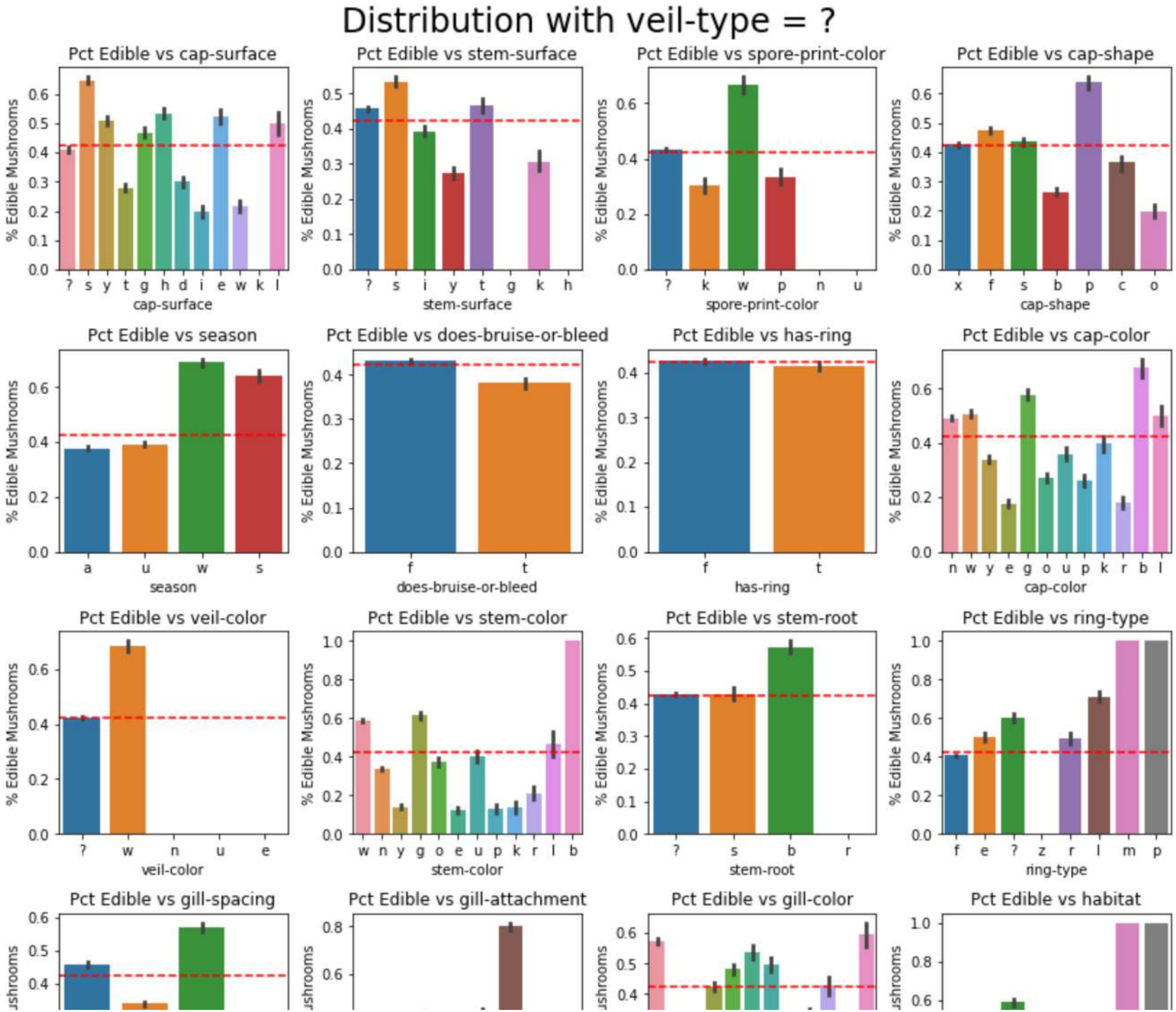


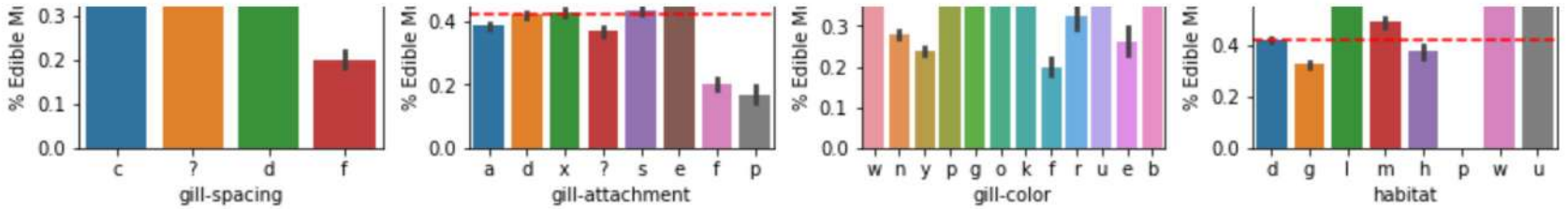


In [21]:

```
X_exploratory = X_train[X_train['veil-type'] == '?']
y_exploratory = y_train[X_train['veil-type'] == '?']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['veil-type'])):
    if label != 'veil-type':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with veil-type = ?", fontsize=24)
fig.tight_layout()
plt.show()
```

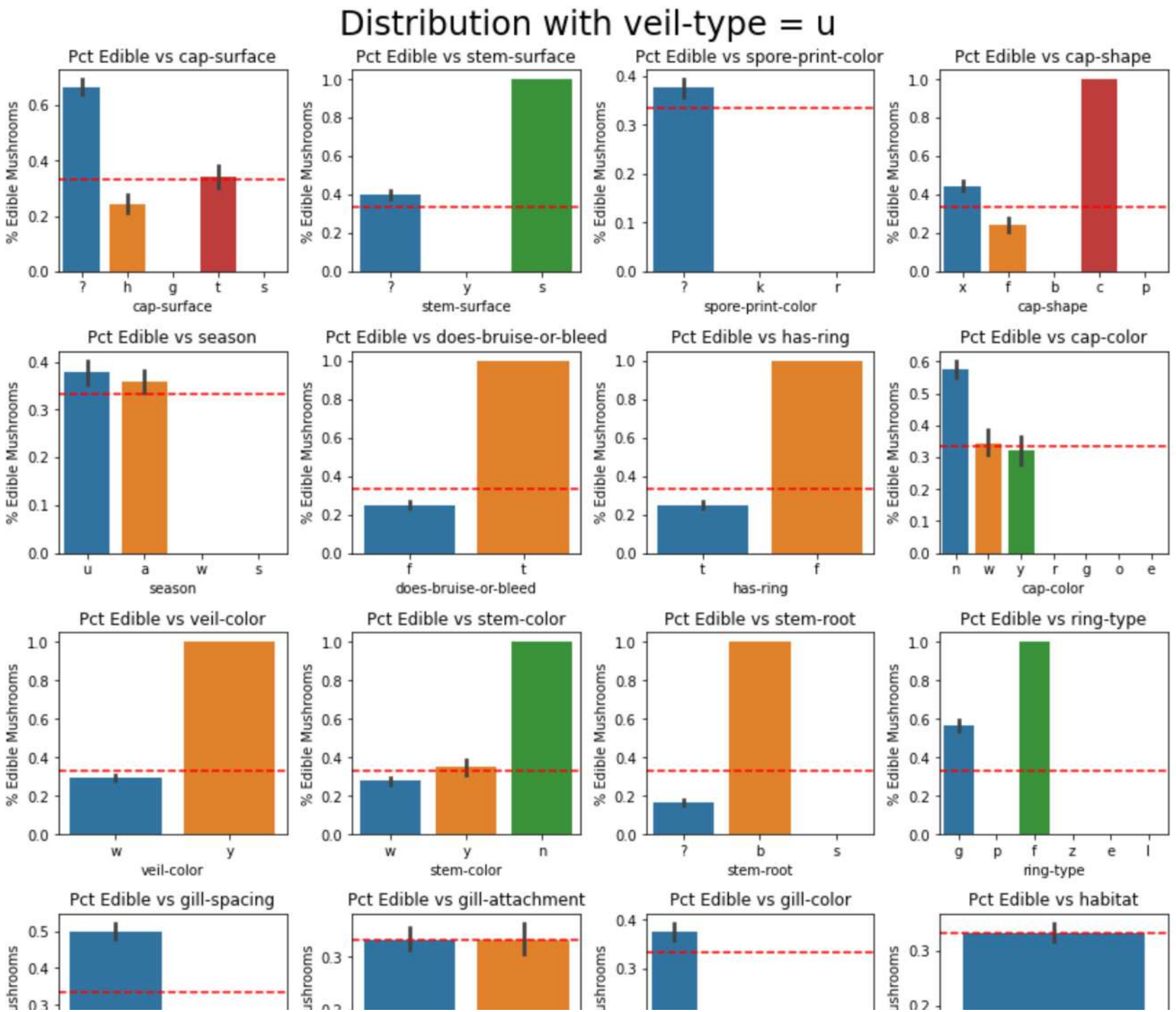

Distribution with veil-type = ?
Pct Edible vs cap-surface
% Edible Mushrooms
cap-surface
? s y t g h d i e w k l
Pct Edible vs stem-surface
stem-surface
? s i y t g k h
Pct Edible vs spore-print-color
spore-print-color
? k w p n u
Pct Edible vs cap-shape
cap-shape
x f s b p c o
Pct Edible vs season
season
a u w s
Pct Edible vs does-bruise-or-bleed
does-bruise-or-bleed
f t
Pct Edible vs has-ring
has-ring
f t
Pct Edible vs cap-color
cap-color
n w y e g o u p k r b l
Pct Edible vs veil-color
veil-color
? w n u e
Pct Edible vs stem-color
stem-color
w n y g o e u p k r l b
Pct Edible vs stem-root
stem-root
? s b r
Pct Edible vs ring-type
ring-type
f e ? z r l m p
Pct Edible vs gill-spacing
Pct Edible vs gill-attachment
Pct Edible vs gill-color
Pct Edible vs habitat

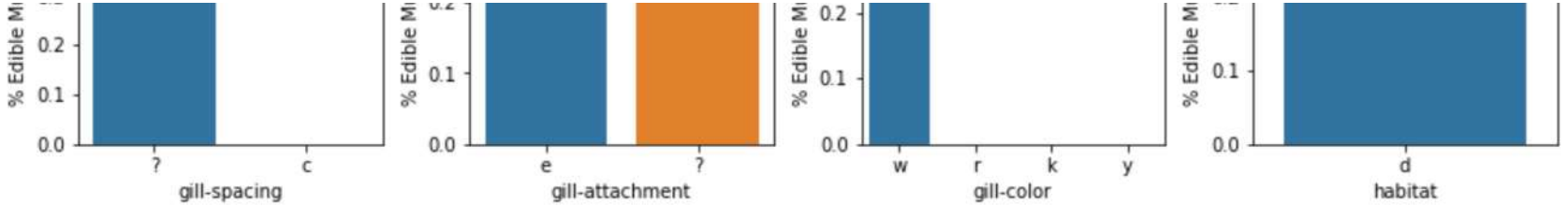


In [22]:

```
X_exploratory = X_train[X_train['veil-type'] == 'u']
y_exploratory = y_train[X_train['veil-type'] == 'u']

fig, ax = plt.subplots(4, 4, figsize=(12, 12))
for i, label in enumerate(set(categorical_feat) - set(['veil-type'])):
    if label != 'veil-type':
        thisax = ax[i//4][i%4]
        # Order by counts for easier visualization
        order = X_exploratory[label].value_counts().sort_values(ascending=False).index
        sns.barplot(data=X_exploratory, x=label, y=y_exploratory, order=order, ax=thisax)
        # Mark a line at the average percent for this subset of the training dataset
        thisax.axhline(y=np.mean(y_exploratory), color='red', linestyle='dashed')
        thisax.set_ylabel('% Edible Mushrooms')
        thisax.set_title('Pct Edible vs ' + label)
fig.suptitle("Distribution with veil-type = u", fontsize=24)
fig.tight_layout()
plt.show()
```


Distribution with veil-type = u
Pct Edible vs cap-surface
% Edible Mushrooms
cap-surface
? h g t s
Pct Edible vs stem-surface
stem-surface
? y s
Pct Edible vs spore-print-color
spore-print-color
? k r
Pct Edible vs cap-shape
cap-shape
x f b c p
Pct Edible vs season
season
u a w s
Pct Edible vs does-bruise-or-bleed
does-bruise-or-bleed
f t
Pct Edible vs has-ring
has-ring
t f
Pct Edible vs cap-color
cap-color
n w y r g o e
Pct Edible vs veil-color
veil-color
w y
Pct Edible vs stem-color
stem-color
w y n
Pct Edible vs stem-root
stem-root
? b s
Pct Edible vs ring-type
ring-type
g p f z e l
Pct Edible vs gill-spacing
Pct Edible vs gill-attachment
Pct Edible vs gill-color
Pct Edible vs habitat



There are a large number of differences in distributions splitting over the binary variables. For example, some combinations of variables have all examples in one class or another. Let's take a programmatic approach to identify the most impactful differences:

In [23]:

```
for binary_feat, val in (('has-ring', 't'), ('does-bruise-or-bleed', 't'), ('veil-type', 'u')):
    print('Checking split feature {}:'.format(binary_feat))
    # Split the data into two sets
    X_1 = X_train[X_train[binary_feat] == val]
    y_1 = y_train[X_train[binary_feat] == val]
    X_2 = X_train[X_train[binary_feat] != val]
    y_2 = y_train[X_train[binary_feat] != val]
    # Look for differences in means
    for label in categorical_feat:
        zeros = [0, 0]
        ones = [0, 0]
        statements = []
        for cat in set(X_1[label].unique()) & set(X_2[label].unique()): # Only check categories appearing in both sus
            mean1 = np.mean(y_1[X_1[label] == cat])
            mean2 = np.mean(y_2[X_2[label] == cat])
            num1 = np.sum(X_1[label] == cat)
            num2 = np.sum(X_2[label] == cat)
            if mean1 != mean2:
                if mean1 == 1.0:
                    ones[0] += num1
                elif mean1 == 0.0:
                    zeros[0] += num1
                if mean2 == 1.0:
                    ones[1] += num2
                elif mean2 == 0.0:
                    zeros[1] += num2
                # Filter
                #if ((mean1 == 1.0 and mean2 < 0.5) or (mean1 == 0.0 and mean2 > 0.5)):
                if mean1 == 1.0 or mean1 == 0.0 or mean2 == 1.0 or mean2 == 0.0:
                    statements.append("Feature {} category {} has means {:.03f} ({}) and {:.03f} ({})".format(
                        label, cat, mean1, num1, mean2, num2))
                #elif ((mean2 == 1.0 and mean1 < 0.5) or (mean2 == 0.0 and mean1 > 0.5)):
                #    statements.append("Feature {} category {} has means {:.03f} ({}) and {:.03f} ({})".format(
                #        label, cat, mean1, num1, mean2, num2))
        if zeros[0] > 1000 or zeros[1] > 1000 or ones[0] > 1000 or ones[1] > 1000:
            for stmt in statements:
                print(stmt)
            print("Totals for feature {}: {} {}".format(label, zeros, ones))
    print() # newline
```

```
Checking split feature has-ring:
Feature cap-shape category o has means 0.000 (353) and 0.271 (969)
Feature cap-shape category s has means 0.000 (869) and 0.514 (4830)
Feature cap-shape category b has means 0.000 (1491) and 0.332 (3793)
Totals for feature cap-shape: [2713, 0] [0, 0]
Feature cap-surface category g has means 0.000 (536) and 0.469 (4188)
Feature cap-surface category i has means 0.792 (466) and 0.000 (1406)
Feature cap-surface category d has means 0.000 (246) and 0.330 (2480)
Totals for feature cap-surface: [782, 1406] [0, 0]
Feature cap-color category p has means 0.000 (457) and 0.360 (1246)
Feature cap-color category r has means 0.000 (629) and 0.239 (828)
Feature cap-color category o has means 0.000 (676) and 0.328 (2324)
Feature cap-color category l has means 0.000 (191) and 0.670 (560)
Totals for feature cap-color: [1953, 0] [0, 0]
Feature gill-attachment category x has means 1.000 (706) and 0.368 (6707)
Feature gill-attachment category p has means 0.000 (353) and 0.290 (497)
Feature gill-attachment category f has means 0.000 (353) and 0.250 (1412)
Feature gill-attachment category s has means 0.000 (706) and 0.500 (4942)
Totals for feature gill-attachment: [1412, 0] [706, 0]
Feature gill-color category r has means 0.000 (128) and 0.323 (1093)
Feature gill-color category y has means 0.000 (707) and 0.264 (5543)
Feature gill-color category o has means 0.000 (546) and 0.704 (1736)
Feature gill-color category f has means 0.000 (353) and 0.250 (1412)
Feature gill-color category u has means 0.000 (110) and 0.480 (913)
Totals for feature gill-color: [1844, 0] [0, 0]
Feature stem-root category b has means 1.000 (1412) and 0.400 (1765)
Totals for feature stem-root: [0, 0] [1412, 0]
Feature ring-type category f has means 0.000 (1765) and 0.431 (37152)
Totals for feature ring-type: [1765, 0] [0, 0]

Checking split feature does-bruise-or-bleed:
Feature gill-attachment category e has means 1.000 (1059) and 0.538 (4589)
Feature gill-attachment category s has means 0.000 (353) and 0.467 (5295)
Feature gill-attachment category a has means 1.000 (353) and 0.368 (11935)
Totals for feature gill-attachment: [353, 0] [1412, 0]
Feature stem-color category p has means 0.000 (353) and 0.196 (672)
Feature stem-color category r has means 0.000 (240) and 0.377 (302)
Feature stem-color category e has means 0.000 (541) and 0.189 (969)
Feature stem-color category u has means 0.000 (134) and 0.449 (1188)
Totals for feature stem-color: [1268, 0] [0, 0]
Feature ring-type category g has means 1.000 (353) and 0.398 (887)
Feature ring-type category p has means 1.000 (195) and 0.000 (717)
Feature ring-type category r has means 1.000 (172) and 0.425 (1227)
Feature ring-type category ? has means 1.000 (353) and 0.500 (1412)
```

```
Totals for feature ring-type: [0, 717] [1073, 0]

Checking split feature veil-type:
Feature gill-spacing category c has means 0.000 (1059) and 0.458 (20827)
Totals for feature gill-spacing: [1059, 0] [0, 0]
Feature stem-surface category s has means 1.000 (353) and 0.534 (5319)
Feature stem-surface category y has means 0.000 (1059) and 0.275 (3175)
Totals for feature stem-surface: [1059, 0] [353, 0]
Feature ring-type category p has means 0.000 (717) and 1.000 (195)
Feature ring-type category f has means 1.000 (353) and 0.406 (38564)
Feature ring-type category e has means 0.000 (317) and 0.500 (2118)
Feature ring-type category l has means 0.000 (197) and 0.713 (1230)
Totals for feature ring-type: [1231, 0] [353, 195]
```

There are several perfect correlations with large enough numbers to be impactful. We'll choose a few interaction terms to create as features.

In [24]:
```
X_train['ring-t-cap-shape-bso'] = ((X_train['has-ring'] == 't') & (X_train['cap-shape'].isin(['b', 's', 'o']))).asty
X_train['ring-f-cap-surface-i'] = ((X_train['has-ring'] == 'f') & (X_train['cap-surface'] == 'i')).astype(np.float64
X_train['ring-t-cap-color-oplr'] = ((X_train['has-ring'] == 't') & (X_train['cap-color'].isin(['o', 'p', 'l', 'r']))
X_train['ring-t-gill-attachment-spf'] = ((X_train['has-ring'] == 't') & (X_train['gill-attachment'].isin(['s', 'p',
X_train['ring-t-gill-color-youfr'] = ((X_train['has-ring'] == 't') & (X_train['gill-color'].isin(['y', 'o', 'u', 'f'
X_train['bruise-t-gill-attachment-ea'] = ((X_train['does-bruise-or-bleed'] == 't') & (X_train['gill-attachment'].isi
X_train['bruise-t-stem-color-eupr'] = ((X_train['does-bruise-or-bleed'] == 't') & (X_train['stem-color'].isin(['e',
X_train['bruise-t-ring-type-prg?'] = ((X_train['does-bruise-or-bleed'] == 't') & (X_train['ring-type'].isin(['p', 'r
X_train['veil-u-gill-spacing-c'] = ((X_train['veil-type'] == 'u') & (X_train['gill-spacing'] == 'c')).astype(np.float
X_train['veil-u-stem-surface-y'] = ((X_train['veil-type'] == 'u') & (X_train['stem-surface'] == 'y')).astype(np.float
X_train['veil-u-ring-epl'] = ((X_train['veil-type'] == 'u') & (X_train['ring-type'].isin(['e', 'p', 'l']))).astype(np

# Repeat for test data
X_test['ring-t-cap-shape-bso'] = ((X_test['has-ring'] == 't') & (X_test['cap-shape'].isin(['b', 's', 'o']))).astype(r
X_test['ring-f-cap-surface-i'] = ((X_test['has-ring'] == 'f') & (X_test['cap-surface'] == 'i')).astype(np.float64)
X_test['ring-t-cap-color-oplr'] = ((X_test['has-ring'] == 't') & (X_test['cap-color'].isin(['o', 'p', 'l', 'r']))).as
X_test['ring-t-gill-attachment-spf'] = ((X_test['has-ring'] == 't') & (X_test['gill-attachment'].isin(['s', 'p', 'f'
X_test['ring-t-gill-color-youfr'] = ((X_test['has-ring'] == 't') & (X_test['gill-color'].isin(['y', 'o', 'u', 'f', 'r
X_test['bruise-t-gill-attachment-ea'] = ((X_test['does-bruise-or-bleed'] == 't') & (X_test['gill-attachment'].isin([
X_test['bruise-t-stem-color-eupr'] = ((X_test['does-bruise-or-bleed'] == 't') & (X_test['stem-color'].isin(['e', 'u'
X_test['bruise-t-ring-type-prg?'] = ((X_test['does-bruise-or-bleed'] == 't') & (X_test['ring-type'].isin(['p', 'r',
X_test['veil-u-gill-spacing-c'] = ((X_test['veil-type'] == 'u') & (X_test['gill-spacing'] == 'c')).astype(np.float64
X_test['veil-u-stem-surface-y'] = ((X_test['veil-type'] == 'u') & (X_test['stem-surface'] == 'y')).astype(np.float64
X_test['veil-u-ring-epl'] = ((X_test['veil-type'] == 'u') & (X_test['ring-type'].isin(['e', 'p', 'l']))).astype(np.fl
```

Now let's explore large shifts in population means that could improve accuracy on a large number of training examples:

In [25]:
```
for binary_feat, val in (('has-ring', 't'), ('does-bruise-or-bleed', 't'), ('veil-type', 'u')):
    print('Checking split feature {}:'.format(binary_feat))
    # Split the data into two sets
    X_1 = X_train[X_train[binary_feat] == val]
    y_1 = y_train[X_train[binary_feat] == val]
    X_2 = X_train[X_train[binary_feat] != val]
    y_2 = y_train[X_train[binary_feat] != val]
    # Look for differences in means
    for label in categorical_feat:
        for cat in set(X_1[label].unique()) & set(X_2[label].unique()): # Only check categories appearing in both sus
            mean1 = np.mean(y_1[X_1[label] == cat])
            mean2 = np.mean(y_2[X_2[label] == cat])
            diff = abs(mean1 - mean2)
            num1 = np.sum(X_1[label] == cat)
            num2 = np.sum(X_2[label] == cat)
            num = min(num1, num2)
            #num = np.sum(df[label] == cat)
            if diff > 0.25 and diff * num > 500: # Threshhold
                print("Feature {} category {} has a difference of means of {:.03f} (num affected: {})".format(
                    label, cat, diff, num))
    print() # newline
```

```
Checking split feature has-ring:
Feature cap-surface category t has a difference of means of 0.312 (num affected: 2229)
Feature gill-attachment category a has a difference of means of 0.262 (num affected: 2816)
Feature stem-root category b has a difference of means of 0.600 (num affected: 1412)
Feature ring-type category f has a difference of means of 0.431 (num affected: 1765)

Checking split feature does-bruise-or-bleed:
Feature cap-shape category s has a difference of means of 0.320 (num affected: 2629)
Feature gill-attachment category d has a difference of means of 0.330 (num affected: 4236)

Checking split feature veil-type:
Feature gill-attachment category e has a difference of means of 0.467 (num affected: 2118)
Feature stem-root category ? has a difference of means of 0.262 (num affected: 2118)
Feature stem-color category w has a difference of means of 0.310 (num affected: 2453)
Feature veil-color category w has a difference of means of 0.390 (num affected: 2061)
```

We'll create another few interaction terms from these splits as well.

In [26]:
```
X_train['ring-t-stem-root-b'] = ((X_train['has-ring'] == 't') & (X_train['stem-root'] == 'b')).astype(np.float64)
X_train['ring-t-ring-type-f'] = ((X_train['has-ring'] == 't') & (X_train['ring-type'] == 'f')).astype(np.float64)
X_train['bruise-f-gill-attachment-d'] = ((X_train['does-bruise-or-bleed'] == 'f') & (X_train['gill-attachment'] == 'd
X_train['veil-?-gill-attachment-e'] = ((X_train['veil-type'] == '?') & (X_train['gill-attachment'] == 'e')).astype(np

# Repeat for test data
X_test['ring-t-stem-root-b'] = ((X_test['has-ring'] == 't') & (X_train['stem-root'] == 'b')).astype(np.float64)
X_test['ring-t-ring-type-f'] = ((X_test['has-ring'] == 't') & (X_train['ring-type'] == 'f')).astype(np.float64)
X_test['bruise-f-gill-attachment-d'] = ((X_test['does-bruise-or-bleed'] == 'f') & (X_test['gill-attachment'] == 'd'))
X_test['veil-?-gill-attachment-e'] = ((X_test['veil-type'] == '?') & (X_test['gill-attachment'] == 'e')).astype(np.fl
```

At this time, we'll also create some new features from the categorical features with mostly null values which we will be dropping in the next section. Some of these features have categories with perfect correlation with the target class, so saving this information could be very valuable.

In [27]:
```
X_train['spore-print-color-nur'] = X_train['spore-print-color'].isin(['n', 'u', 'r']).astype(np.float64)
X_train['veil-color-nue'] = X_train['veil-color'].isin(['n', 'u', 'e']).astype(np.float64)
X_train['stem-surface-gh'] = X_train['stem-surface'].isin(['g', 'h']).astype(np.float64)
X_train['stem-root-r'] = (X_train['stem-root'] == 'r').astype(np.float64)

# Repeat for test data
X_test['spore-print-color-nur'] = X_test['spore-print-color'].isin(['n', 'u', 'r']).astype(np.float64)
X_test['veil-color-nue'] = X_test['veil-color'].isin(['n', 'u', 'e']).astype(np.float64)
X_test['stem-surface-gh'] = X_test['stem-surface'].isin(['g', 'h']).astype(np.float64)
X_test['stem-root-r'] = (X_test['stem-root'] == 'r').astype(np.float64)
```

In [28]:
```
X_train['cap-stem-color-match'] = (X_train['cap-color'] == X_train['stem-color']).astype(np.float64)

X_test['cap-stem-color-match'] = (X_test['cap-color'] == X_test['stem-color']).astype(np.float64)
```

Additional features (complex interaction terms):

In [29]:
```
X_train['add-feature-1'] = ((X_train['does-bruise-or-bleed'] == 't') & (X_train['has-ring'] == 't') &
                            (X_train['cap-color'] == 'n')).astype(np.float64)
X_train['add-feature-2'] = ((X_train['veil-type'] == 'u') & ((X_train['does-bruise-or-bleed'] == 't') |
                            (X_train['has-ring'] == 'f') | (X_train['veil-color'] == 'y'))).astype(np.float64)
X_train['add-feature-3'] = ((X_train['stem-height'] >= 20) | (X_train['cap-diameter'] >= 35)).astype(np.float64)

X_test['add-feature-1'] = ((X_test['does-bruise-or-bleed'] == 't') & (X_test['has-ring'] == 't') &
                           (X_test['cap-color'] == 'n')).astype(np.float64)
X_test['add-feature-2'] = ((X_test['veil-type'] == 'u') & ((X_test['does-bruise-or-bleed'] == 't') |
                           (X_test['has-ring'] == 'f') | (X_test['veil-color'] == 'y'))).astype(np.float64)
X_test['add-feature-3'] = ((X_test['stem-height'] >= 20) | (X_test['cap-diameter'] >= 35)).astype(np.float64)
```

In [30]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(50213, 43)
(10856, 43)
```

In [31]:
```
new_features = ['ring-t-cap-shape-bso', 'ring-f-cap-surface-i', 'ring-t-cap-color-oplr', 'ring-t-gill-attachment-spf',
                'ring-t-gill-color-youfr', 'bruise-t-gill-attachment-ea', 'bruise-t-stem-color-eupr',
                'bruise-t-ring-type-prg?', 'veil-u-gill-spacing-c', 'veil-u-stem-surface-y', 'veil-u-ring-epl',
                'ring-t-stem-root-b', 'ring-t-ring-type-f', 'bruise-f-gill-attachment-d', 'veil-?-gill-attachment-e',
                'spore-print-color-nur', 'veil-color-nue', 'stem-surface-gh', 'stem-root-r', 'cap-stem-color-match',
                'add-feature-1', 'add-feature-2', 'add-feature-3']
```

4. Data Processing

We now have to choose what to do with null values. For features with a majority of null values, we can just drop the feature entirely. For other features, we choose to either impute as the mode or as the value corresponding to none ('f'). Since we already converted the null values to '?' for EDA, we'll perform imputation manually instead of using a pipeline like `SimpleImputer`.

In [32]:
```
# Drop
#X_train.drop(['stem-root', 'stem-surface', 'veil-color', 'spore-print-color'], axis=1, inplace=True)
#X_test.drop(['stem-root', 'stem-surface', 'veil-color', 'spore-print-color'], axis=1, inplace=True)

# Impute as mode
X_train.loc[X_train['cap-surface'] == '?', 'cap-surface'] = \
    X_train[X_train['cap-surface'] != '?']['cap-surface'].value_counts().index[0]
X_test.loc[X_test['cap-surface'] == '?', 'cap-surface'] = \
    X_train[X_train['cap-surface'] != '?']['cap-surface'].value_counts().index[0]

# Impute as none
for label in ('gill-attachment', 'gill-spacing', 'ring-type'):
    X_train.loc[X_train[label] == '?', label] = 'f'
    X_test.loc[X_test[label] == '?', label] = 'f'

categorical_feat = ['cap-shape', 'cap-surface', 'cap-color', 'does-bruise-or-bleed',
       'gill-attachment', 'gill-spacing', 'gill-color', 'stem-color', 'veil-type',
       'has-ring', 'ring-type', 'habitat', 'season']
```

We'll use the standard pipeline that we've used throughout the class:

In [33]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

pipeline = ColumnTransformer(transformers=[
    ("drop", "drop", ['stem-root', 'stem-surface', 'veil-color', 'spore-print-color']),
    ("scale", StandardScaler(), numerical_feat),
    ("encode", OneHotEncoder(
        drop=['x', 's', 'n', 'f', 'f', 'f', 'w', 'w', '?', 'f', 'f', 'd', 'a'],
        handle_unknown='ignore'
    ), categorical_feat), # drop the most common element in each category (or NaN if present)
    ("passthrough", "passthrough", new_features)
], remainder="drop", sparse_threshold=0)

X_train_processed = pipeline.fit_transform(X_train)
X_train_processed.shape
```

Out[33]:
```
(50213, 103)
```

In this case, the pipeline is simple. We one-hot encode categorical features and standard scale numerical features. We decline to standard scale binary features in order to make the interpretation of coefficients in some of our models easier.

This pipeline results in >100 features, which is quite a lot.

In [34]:
```
X_test_processed = pipeline.transform(X_test)
X_test_processed.shape
```

```
C:\Users\mcrai\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\LocalCache\local-packages\P
ython310\site-packages\sklearn\preprocessing\_encoders.py:170: UserWarning: Found unknown categories in columns [7]
during transform. These unknown categories will be encoded as all zeros
  warnings.warn(
```

Out[34]:
```
(10856, 103)
```

Note that several values in the test data do not appear in the training data. We choose to ignore these categories, since with no training data we cannot reliably estimate coefficients for these.

In [35]:
```
for label in categorical_feat:
    if set(X_test[label].unique()) - set(X_train[label].unique()):
        print(label + ':')
        print(X_train[label].unique())
        print(X_test[label].unique())
```

```
stem-color:
['w' 'y' 'n' 'u' 'b' 'l' 'r' 'p' 'e' 'k' 'g' 'o']
['g' 'n' 'w' 'e' 'y' 'u' 'k' 'o' 'f']
```

In order to make coefficient testing feasible, we will select only a subset of features. Based on the EDA above, we can drop categorical features which do not add very much information, which have a large number of null values, or which have too many categories with relatively few examples.

In [36]:
```
subset_pipeline = ColumnTransformer(transformers=[
    ("scale", StandardScaler(), numerical_feat),
    ("encode", OneHotEncoder(
        drop=['f', 'f', 'w', '?', 'a'],
        handle_unknown='ignore'
    ), ['gill-attachment', 'gill-spacing', 'gill-color', 'veil-type', 'season'])
], remainder="drop", sparse_threshold=0)

Xs_train_processed = subset_pipeline.fit_transform(X_train)
Xs_train_processed.shape
```

Out[36]:
```
(50213, 26)
```

In [37]:
```
Xs_test_processed = subset_pipeline.transform(X_test)
Xs_test_processed.shape
```

Out[37]:
```
(10856, 26)
```

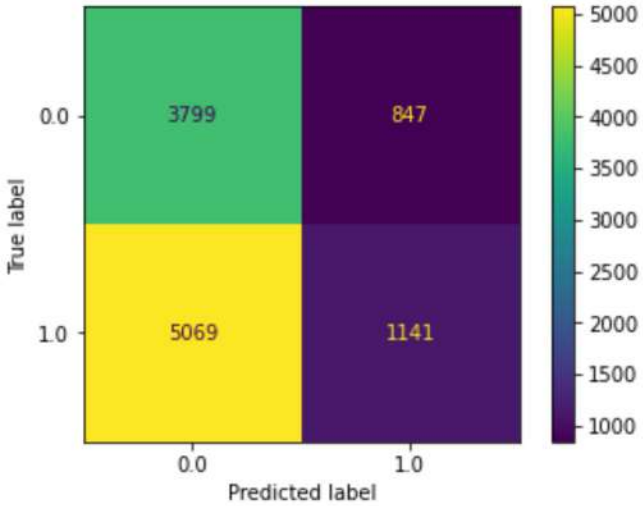
6. Logistic Regression & Statistical Hypothesis Testing

In [41]:
```
import statsmodels.api as sm
from statsmodels.tools import add_constant

glm = sm.GLM(endog=y_train, exog=add_constant(Xs_train_processed), family=sm.families.Binomial())
#model = glm.fit_regularized(method='elastic_net', alpha=1.0, maxiter=1000)
lr_model = glm.fit(maxiter=1000)
```

In [42]:
```
from sklearn.metrics import ConfusionMatrixDisplay

y_prob = lr_model.predict(add_constant(Xs_test_processed))
y_pred = (y_prob > 0.5).astype(np.float64)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred);
```



In [43]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

print("Accuracy: {}".format(accuracy_score(y_test, y_pred)))
print("Precision: {}".format(precision_score(y_test, y_pred)))
print("Recall: {}".format(recall_score(y_test, y_pred)))
print("F-1: {}".format(f1_score(y_test, y_pred)))
```

```
Accuracy: 0.45504789977892407
Precision: 0.573943661971831
Recall: 0.18373590982286633
F-1: 0.27836057575018297
```

The performance of the Logistic Regression classifier is not great (45%). With less than 50% accuracy, we would be better off choosing the opposite of whatever the classifier predicts, to achieve an accuracy of 55%.

In [44]:
```
lr_model.summary()
```

Out[44]:

Generalized Linear Model Regression Results

Dep. Variable:	y	**No. Observations:**	50213
Model:	GLM	**Df Residuals:**	50186
Model Family:	Binomial	**Df Model:**	26
Link Function:	Logit	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-29021.
Date:	Fri, 03 Mar 2023	**Deviance:**	58043.
Time:	10:00:17	**Pearson chi2:**	4.96e+04
No. Iterations:	6	**Pseudo R-squ. (CS):**	0.1838
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0451	0.038	-1.180	0.238	-0.120	0.030
x1	0.4613	0.022	21.177	0.000	0.419	0.504
x2	-0.1402	0.014	-10.139	0.000	-0.167	-0.113
x3	0.0681	0.019	3.551	0.000	0.030	0.106
x4	-0.2412	0.036	-6.642	0.000	-0.312	-0.170
x5	-0.5371	0.043	-12.575	0.000	-0.621	-0.453
x6	1.2917	0.046	28.058	0.000	1.201	1.382
x7	-2.9624	0.162	-18.276	0.000	-3.280	-2.645
x8	-0.1811	0.044	-4.157	0.000	-0.267	-0.096
x9	-0.1498	0.040	-3.749	0.000	-0.228	-0.071
x10	0.3231	0.024	13.296	0.000	0.275	0.371
x11	0.8741	0.032	27.127	0.000	0.811	0.937
x12	-0.1929	0.091	-2.120	0.034	-0.371	-0.015
x13	-1.0675	0.095	-11.274	0.000	-1.253	-0.882

x14	-2.0795	0.085	-24.609	0.000	-2.245	-1.914
x15	-0.2930	0.042	-7.029	0.000	-0.375	-0.211
x16	-0.5608	0.058	-9.696	0.000	-0.674	-0.447
x17	-1.2723	0.034	-37.930	0.000	-1.338	-1.207
x18	0.3183	0.051	6.272	0.000	0.219	0.418
x19	-0.6275	0.035	-17.697	0.000	-0.697	-0.558
x20	-0.8553	0.069	-12.331	0.000	-0.991	-0.719
x21	-0.5488	0.069	-7.942	0.000	-0.684	-0.413
x22	-1.4373	0.038	-38.151	0.000	-1.511	-1.364
x23	-1.8474	0.054	-34.006	0.000	-1.954	-1.741
x24	1.5397	0.054	28.579	0.000	1.434	1.645
x25	0.0715	0.022	3.253	0.001	0.028	0.115
x26	1.0668	0.038	27.712	0.000	0.991	1.142

Interpretation

Other than the constant, we can see that all of the selected features are statistically significant at the 5% level. The first three coefficients (x1 to x3) are the three numerical features. Cap diameter has a strongly positive value, which means that larger values mean more likely to be edible. This matches the interpretation of the EDA we performed above.

7. Dimensionality Reduction using PCA

In [45]:
```
# PCA: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
# To address the large number of features, let's perform PCA
from sklearn.decomposition import PCA

pca = PCA(n_components=103)
pca.fit(X_train_processed)
np.cumsum(pca.explained_variance_ratio_)
```

Out[45]:
```
array([0.2446768 , 0.33382236, 0.3737641 , 0.40903282, 0.44303185,
       0.47438734, 0.50347754, 0.52999002, 0.55547023, 0.57916046,
       0.6008192 , 0.62007855, 0.63789824, 0.65503509, 0.67122184,
       0.68616891, 0.70022413, 0.71399638, 0.72665249, 0.73873916,
       0.75013603, 0.76111227, 0.77180552, 0.78145879, 0.79098202,
       0.79989183, 0.80842116, 0.81654695, 0.82456065, 0.83225267,
       0.83962711, 0.84660823, 0.85313284, 0.85924358, 0.86529981,
       0.87105781, 0.8766925 , 0.88202696, 0.88717595, 0.89222056,
       0.89709972, 0.90158975, 0.90603185, 0.91031666, 0.91440379,
       0.91835222, 0.92216223, 0.92566989, 0.92912099, 0.93244076,
       0.93561347, 0.93863987, 0.94159281, 0.94448048, 0.9472023 ,
       0.94987781, 0.95243078, 0.95492123, 0.95724893, 0.95950741,
       0.96169386, 0.96377785, 0.96581411, 0.96768827, 0.96953581,
       0.97134655, 0.97311166, 0.97481475, 0.97638917, 0.97794389,
       0.97940959, 0.98075674, 0.98207102, 0.98331474, 0.98448667,
       0.98560605, 0.98670927, 0.98777367, 0.9887861 , 0.98972539,
       0.99062025, 0.99144869, 0.99223154, 0.99297477, 0.99368417,
       0.99429318, 0.99487101, 0.99540783, 0.99591887, 0.99639049,
       0.99684683, 0.99728271, 0.99767353, 0.99802993, 0.99835282,
       0.99865054, 0.99893332, 0.99920379, 0.99942791, 0.99962901,
       0.99979017, 0.99992859, 1.        ])
```

In [46]:
```
pca.singular_values_
```

Out[46]:
```
array([330.44890913, 199.4608982 , 133.51235246, 125.45927445,
       123.18027172, 118.29455051, 113.94132912, 108.77600464,
       106.6373749 , 102.82354143,  98.31605326,  92.71047233,
        89.17806215,  87.45274108,  84.99390311,  81.67438602,
        79.20030154,  78.39896035,  75.15504114,  73.44483006,
        71.31824305,  69.98979286,  69.08167292,  65.63644112,
        65.19286556,  63.05829706,  61.69719294,  60.22000055,
        59.80318159,  58.59063408,  57.36835498,  55.81751237,
        53.96165954,  52.22216265,  51.9887249 ,  50.69249124,
        50.14678319,  48.79249488,  47.93677683,  47.44841879,
        46.66387235,  44.76437982,  44.52478871,  43.72939597,
        42.70880083,  41.97783092,  41.23551533,  39.56549921,
        39.24516995,  38.49126602,  37.62901607,  36.75119037,
        36.30239429,  35.8989241 ,  34.85278162,  34.55504749,
        33.75440104,  33.33857471,  32.23087615,  31.74794862,
        31.23757845,  30.49695973,  30.14562418,  28.920845  ,
        28.71473686,  28.42733229,  28.06681619,  27.56939131,
        26.50747711,  26.34107168,  25.57591485,  24.51972977,
        24.21879296,  23.55959995,  22.8696633 ,  22.35093756,
        22.18908094,  21.79517389,  21.25645571,  20.4742353 ,
        19.98409109,  19.22827111,  18.69156924,  18.21256462,
        17.79318179,  16.48612979,  16.05861985,  15.47826409,
        15.10207568,  14.50788179,  14.27086854,  13.9473184 ,
        13.20681215,  12.61182166,  12.00427756,  11.52679753,
        11.23402423,  10.98669396,  10.00110215,   9.47360458,
         8.48080372,   7.85952758,   5.64545163])
```

Based on the cumulative explained variance, we can see that the first 26 components capture 80% of the variance in the data. This aligns with a common "rule of thumb" found in the literature, and is a much more reasonable number of features for our models to train on.

In [47]:
```
pca_final = PCA(n_components=27)
X_train_pca = pca_final.fit_transform(X_train_processed)
X_test_pca = pca_final.transform(X_test_processed)
```

8. Experiment with any 2 other models (Non-Ensemble)

Let's try a number of different unoptimized classifiers and see how they perform:

Support Vector Machine

In [48]:
```
# Models: https://scikit-learn.org/stable/supervised_learning.html
from sklearn.svm import SVC

svc = SVC(kernel='rbf') # RBF kernel (default)
#model_svc = svc.fit(X_train_pca, y_train)
```

In [49]:
```
from sklearn.model_selection import cross_validate

# PCA
scores = cross_validate(svc, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5812666716151815
test_precision: 0.5179938758126964
test_recall: 0.511513240721321
test_f1: 0.5063157358153766
```

In [50]:
```
# Processed data
scores = cross_validate(svc, X_train_processed, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_j
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5516132492461265
test_precision: 0.48422503422629093
test_recall: 0.45763078306429017
test_f1: 0.45738680193812353
```

K-Nearest Neighbors

In [51]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)
```

In [52]:
```
# PCA
scores = cross_validate(knn, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5099890189373505
test_precision: 0.44146938601288904
test_recall: 0.6119384466031559
test_f1: 0.5108734633092956
```

In [53]:
```
# Processed data
scores = cross_validate(knn, X_train_processed, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_j
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.472090282646996
test_precision: 0.4082622478591632
test_recall: 0.5996367817581505
test_f1: 0.4844455678257016
```

Naive Bayesian Classifier

In [54]:
```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
```

In [55]:
```
# PCA
scores = cross_validate(gnb, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5564718910437756
test_precision: 0.46724877103679524
test_recall: 0.4462846691141156
test_f1: 0.4496582619793704
```

Neural Network

In [56]:
```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(100, 50))
```

In [57]:
```
# PCA
scores = cross_validate(mlp, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5484856899835785
test_precision: 0.46606810203615157
test_recall: 0.5262028904451163
test_f1: 0.49080025309724995
```

In [58]:
```
# Processed data
scores = cross_validate(mlp, X_train_processed, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_j
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5751120946708679
test_precision: 0.503854771890021
test_recall: 0.4775650895796992
test_f1: 0.4754800914328555
```

The models generally perform better on the PCA transformed data rather than the higher-dimensional encoded data. Of the four models, SVM looks the most promising. The Naive Bayes and KNN classifiers perform well, but since those methods have relatively few hyperparameters to tune, we will choose the MLP since it can likely achieve a higher accuracy after optimization.

9. Experiment with 1 Ensemble Method

In [59]:
```
# Ensemble Methods: https://scikit-learn.org/stable/modules/ensemble.html
from sklearn.ensemble import GradientBoostingClassifier

gbc = GradientBoostingClassifier()
```

In [60]:
```
# PCA
scores = cross_validate(gbc, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.578238949930158
test_precision: 0.49829098403644717
test_recall: 0.49740746614011844
test_f1: 0.492295088724127
```

In [61]:
```
# Processed data
scores = cross_validate(gbc, X_train_processed, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_j
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5636015997571724
test_precision: 0.4882006968419814
test_recall: 0.4968794173866634
test_f1: 0.4850762600151999
```

In [62]:
```
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier()
```

In [63]:
```
# PCA
scores = cross_validate(rfc, X_train_pca, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_jobs=-1
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5834184704634838
test_precision: 0.5209188991627871
test_recall: 0.46964883712074057
test_f1: 0.48603017665097603
```

In [64]:
```
# Processed data
scores = cross_validate(rfc, X_train_processed, y_train, scoring=('accuracy', 'precision', 'recall', 'f1'), cv=5, n_j
for k, v in scores.items():
    if k.startswith("test_"):
        print("{}: {}".format(k, np.mean(v)))
```

```
test_accuracy: 0.5853285602044649
test_precision: 0.5049361556412636
test_recall: 0.5309216242284938
test_f1: 0.5094528851109003
```

The ensemble methods looks promising. The both the gradient-boosted decision tree classifier and the random forest classifier have nearly 57% accuracy on held-out data already.

10. Cross-Validation & Hyperparameter Tuning for All 3 Models

In [64]:
```
# Cross-Validation: https://scikit-learn.org/stable/modules/cross_validation.html
# Hyperparameter Tuning: https://scikit-learn.org/stable/modules/grid_search.html
from sklearn.model_selection import GridSearchCV

svm_params = {
    "kernel": ["rbf", "sigmoid"],
    "C": [0.01, 0.25, 0.5, 1]
}

best_svm = GridSearchCV(SVC(), svm_params, scoring='accuracy', n_jobs=-1, cv=10).fit(X_train_pca, y_train)
print(best_svm.best_score_)
print(best_svm.best_params_)
```

```
0.6034727700012
{'C': 0.5, 'kernel': 'rbf'}
```

In [68]:
```
mlp_params = {
    "hidden_layer_sizes": [(100, 50), (50, 30), (64, 64, 32)],
    "alpha": [1e-3, 1e-4],
    "max_iter": [200, 100],
    "activation": ["relu", "tanh"]
}

best_mlp = GridSearchCV(MLPClassifier(), mlp_params, scoring='accuracy', n_jobs=-1, cv=10).fit(X_train_pca, y_train)
print(best_mlp.best_score_)
print(best_mlp.best_params_)
```

```
0.6295210137335576
{'activation': 'tanh', 'alpha': 0.001, 'hidden_layer_sizes': (50, 30), 'max_iter': 200}
```

In [69]:
```
gb_params = {
    "n_estimators": [100, 150, 200, 300],
    "subsample": [1.0, 0.9, 0.8],
    "min_samples_split": [2, 100, 200, 400],
    "max_depth": [3, 5, 7],
    "max_features": [None, "sqrt"]
}

best_gb = GridSearchCV(GradientBoostingClassifier(), gb_params, scoring='accuracy', n_jobs=-1, cv=10).fit(X_train_pca
print(best_gb.best_score_)
print(best_gb.best_params_)
```

```
0.629143328803573
{'max_depth': 5, 'max_features': 'sqrt', 'min_samples_split': 200, 'n_estimators': 300, 'subsample': 0.8}
```

In [87]:
```
rf_params = {
    "n_estimators": [150, 200, 250],
    "criterion": ["gini", "entropy"],
    "min_samples_split": [50, 100, 200],
    "max_depth": [1, 3, 5, 7],
    "max_features": ["sqrt", None],
    "oob_score": [True, False]
}

best_rf = GridSearchCV(RandomForestClassifier(), rf_params, scoring='accuracy', n_jobs=-1, cv=3).fit(X_train_pca, y_t
print(best_rf.best_score_)
print(best_rf.best_params_)
```

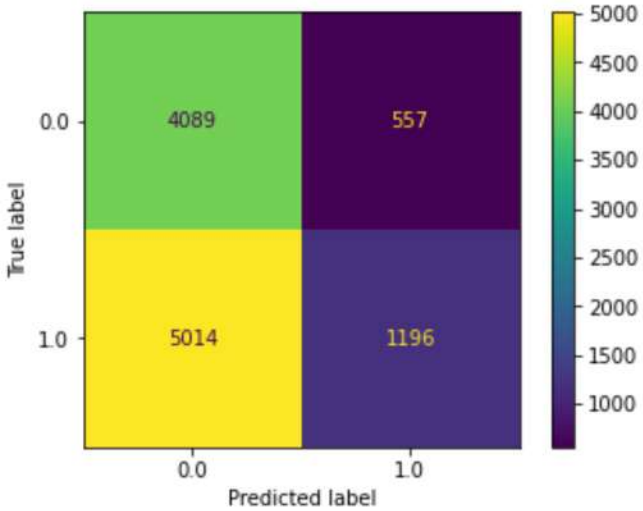
```
0.574991230875939
{'criterion': 'gini', 'max_depth': 5, 'max_features': 'sqrt', 'min_samples_split': 50, 'n_estimators': 250, 'oob_sco
re': True}
```

11. Report Final Results

SVM

In [92]:
```
svc = SVC(kernel='rbf', C=0.5)
model_svc = svc.fit(X_train_pca, y_train)
y_pred = model_svc.predict(X_test_pca)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred);
print("Accuracy: {}".format(accuracy_score(y_test, y_pred)))
print("Precision: {}".format(precision_score(y_test, y_pred)))
print("Recall: {}".format(recall_score(y_test, y_pred)))
print("F-1: {}".format(f1_score(y_test, y_pred)))
```

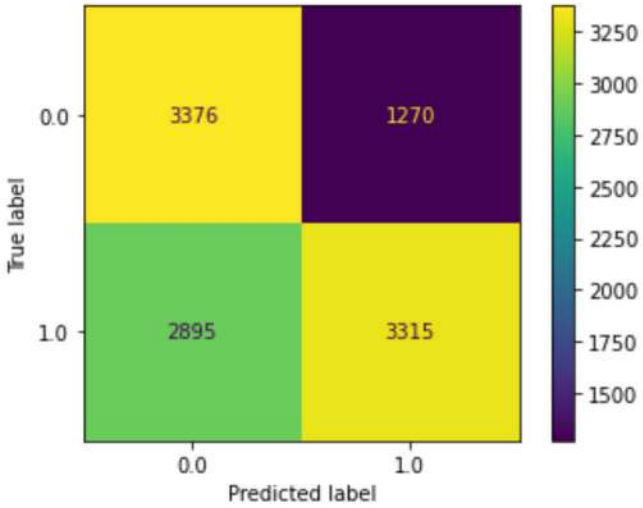
```
Accuracy: 0.48682756079587325
Precision: 0.6822589845978323
Recall: 0.1925925925925926
F-1: 0.3003893005148814
```



MLP Neural Network

In [100…
```
mlp = MLPClassifier(activation='tanh', alpha=1e-3, hidden_layer_sizes=(50, 30), max_iter=200)
model_mlp = mlp.fit(X_train_pca, y_train)
y_pred = model_mlp.predict(X_test_pca)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred);
print("Accuracy: {}".format(accuracy_score(y_test, y_pred)))
print("Precision: {}".format(precision_score(y_test, y_pred)))
print("Recall: {}".format(recall_score(y_test, y_pred)))
print("F-1: {}".format(f1_score(y_test, y_pred)))
```

```
Accuracy: 0.6163411938098747
Precision: 0.7230098146128681
Recall: 0.533816425120773
F-1: 0.6141732283464566
```

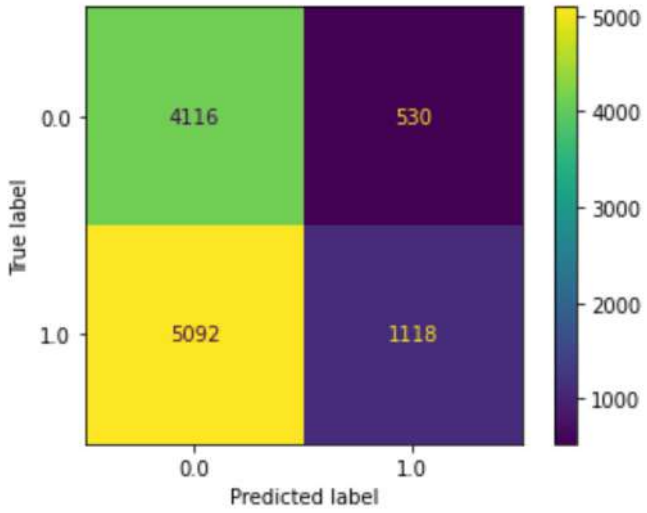


Gradient Boosting Classifier

In [101…

```
gbc = GradientBoostingClassifier(max_depth=5, max_features='sqrt', min_samples_split=200, n_estimators=300, subsample
model_gbc = gbc.fit(X_train_pca, y_train)
y_pred = model_gbc.predict(X_test_pca)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred);
print("Accuracy: {}".format(accuracy_score(y_test, y_pred)))
print("Precision: {}".format(precision_score(y_test, y_pred)))
print("Recall: {}".format(recall_score(y_test, y_pred)))
print("F-1: {}".format(f1_score(y_test, y_pred)))
```

```
Accuracy: 0.4821296978629329
Precision: 0.6783980582524272
Recall: 0.18003220611916265
F-1: 0.2845507762789514
```


4116
530
5092
1118
0.0
1.0
True label
0.0
1.0
Predicted label
5000
4000
3000
2000
1000