Classification de la base Mushroom

Projet RCP209 - Apprentissage statistique 2

Etudiant : Rémi GISCLON

Table des matières

1. Présentation de la base

La base de données analysée provient de The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf.

Cet ensemble de données comprend des descriptions d'échantillons correspondant à 23 espèces de champignons branchiaux de la famille Agaricus et Lepiota (pages 500 à 525). Chaque espèce est identifiée comme définitivement comestible, définitivement vénéneuse, ou de comestibilité inconnue et déconseillée. Cette dernière classe a été combinée avec celle vénéneuse. Le Guide indique clairement qu'il n'y a pas de règle simple pour déterminer la comestibilité d'un champignon.

Le but de cette analyse est de trouver s'il existe un modèle qui permet, contrairement à ce qu'avance le Audubon Society Field Guide to North American Mushrooms, de déterminer la comestibilité d'un champignon.

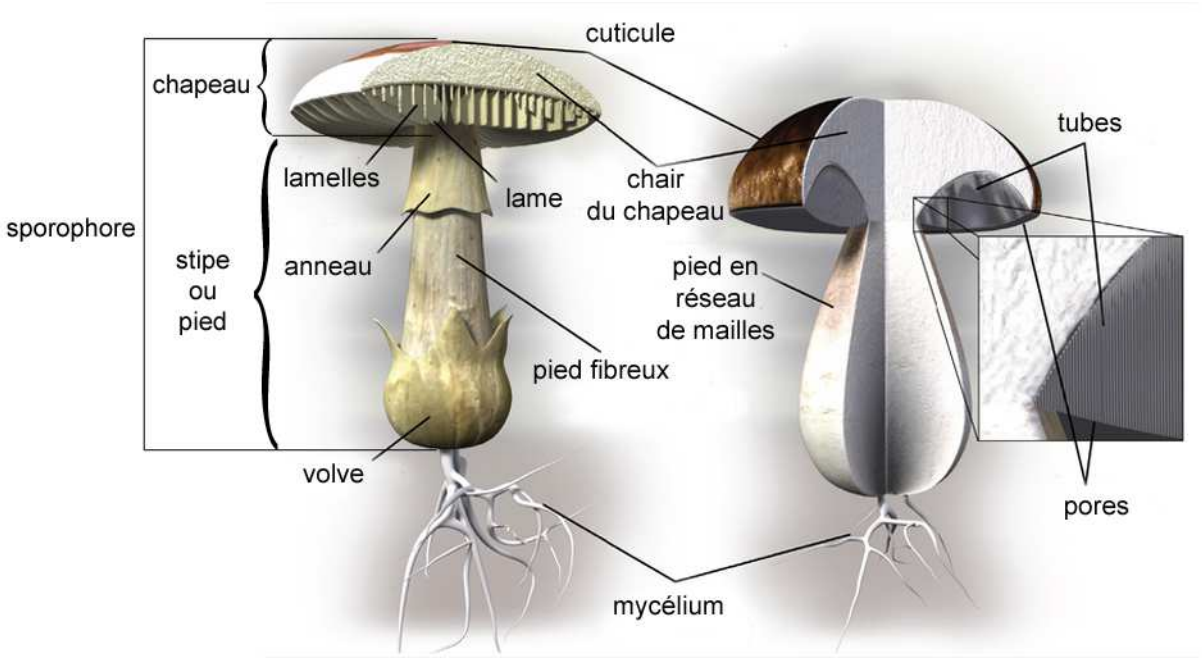
Les champignons sont des végétaux sans feuilles, formés généralement d'un pied surmonté d'un chapeau, à nombreuses espèces comestibles ou vénéneuses. *(Définitions proposées par le Dictionnaire Le Robert).*

La base de données contient **8,125 individus** avec **23 attributs** :

Nom	Description	Valeurs
class	Comestibilité	e = Edible, p = poisonous
cap-shape	La forme du chapeau	b= bell, c = conical, x = convex, f = flat, k = knobbed, s = sunken
cap-surface	Texture du chapeau	f = fibrous, g = grooves, y = scaly, s = smooth
cap-color	La couleur du chapeau	n = brown, b = buff, c = cinnamon, g = gray, r = green, p = pink, u = purple, e = red, w = white, y = yellow
bruises	Bleuissent au toucher	t = yes, f = no
odor	Odeur	a = almond, l = anise, c = creosote, y = fishy, f=foul, m = musty, n = none, p = pungent, s = spicy
gill-attachment	Position des lames	a= attached, d = descending,f = free, n = notched
gill-spacing:	Espacement entre chaque lame	c = close,w = crowded, d = distant
gill-size	Taille des lames	b = broad, n = narrow
gill-color	Couleur des lames	k = black, n = brown, b = buff, h = chocolate, g = gray, r = green, o = orange, p = pink, u = purple, e = red, w = white, y = yellow
stalk-shape	Forme du pied	e = enlarging, t = tapering
stalk-root	Racine du champignon	b = bulbous, c = club, u = cup, e = equal, z = rhizomorphs, r = rooted, ? = missing
stalk-surface-above-ring	Texture du pied au niveau de l'anneau	f = fibrous, y = scaly, k = silky, s = smooth
stalk-surface-below-ring	Texture du pied au niveau de l'anneau	f = fibrous, y = scaly, k = silky, s = smooth
stalk-color-above-ring	Couleur du pied au niveau de l'anneau	n = brown, b = buff, c = cinnamon, g = gray, o = orange, p = pink, e = red, w = white, y = yellow
stalk-color-below-ring	Couleur du pied au niveau sans l'anneau	n = brown, b = buff, c = cinnamon, g = gray, o = orange, p = pink, e = red, w = white, y = yellow
veil-type	Type de voile	p = partial, u = universal
veil-color	Forme du voile	n = brown, o = orange, w = white, y = yellow
ring-number	Nombre d'anneaux	n= none, o = one, t = two
ring-type	Type d'anneaux	c = cobwebby, e = evanescent, f = flaring, l = large, n = none, p = pendant, s = sheathing, z = zone
spore-print-color	Couleur des spores	k = black, n = brown, b = buff, h = chocolate, r = green, o = orange, u = purple, w = white, y = yellow
population	Population du champignon	a = abundant, c = clustered, n = numerous, s = scattered, v = several, y = solitary

Nom	Description	Valeurs
habitat	Environnement du champignon	g = grasses, l = leaves, m = meadows, p = paths, u = urban, w = waste, d = woods

Pour une meilleure compréhension de ces attributs, voici un image explicative tirée de l'article Wikipédia [Chapeau (champignon)]



2. Analyse statistique de la base et modification

a. Analyse univariée

Tous les attributs de la base de données sont des valeurs qualitatives. La plupart des données ne présente pas de notion de distance. Les exceptions sont gill-spacing et population. Cependant ,au vu des résultats des méthodes d'analyse sans tenir en compte ces distances, nous n'avons pas pris la peine d'intégrer cette notion.

Un seul des attributs présente des valeurs manquantes : stalk-root. Cependant il n'est pas clair si la valeur missing se rapporte au fait que l'information est absente ou que le champignon ne présente pas de racine. Nous avons donc gardé cet attribut tel que dans la base initiale et considéré missing comme une des valeurs possibles.

Code Python de l'analyse univariée :

```
#Analyse univariée

C_cols = champi.columns.to_list()
for attribut in C_cols[:]:
    plt.figure()
    sns.countplot(x=attribut , data=champi)
    plt.show()
    print("% des differentes valeurs:")
    print(round((champi[attribut].value_counts()/champi.shape[0]),4)*100)
```

Vous pouvez trouver les graphiques liés à cette analyse en annexe.

La quasi-totalité des attributs ont toutes leurs valeurs exprimées, même si certains ont une valeurs sur-représentée par rapport aux autres, comme la couleur du voile (veil color) ou la texture du chapeau. Les tableaux parlent d'eux même.

Cependant, l'attribut veil-type va être supprimé. Celui-ci peut prendre deux valeurs mais dans la base de données une seule des valeurs est exprimée. Cette information n'est donc d'aucune utilité.

La nouvelle base de données, agaricus-lepiota V2, ne présente plus cet attribut.

b. Analyse bivariée

Code Python de l'analyse bivariée :

```
#Analyse bivariée des attributs par rapport a la classe

for attribut in C_cols[1:]:
    plt.figure(figsize=(30,20))
    plt.subplot(3,3,3)
    sns.countplot(x=attribut ,hue='class', data=champi)
    plt.xlabel(attribut, fontsize=30)
    plt.legend(loc='upper right')
    plt.show()
    print(pd.pivot_table(champi, index=[attribut,"class"], aggfunc = {attribut:np.count_nonzero}))
```

Vous pouvez trouver les graphiques liés à cette analyse en annexe.

Certaines valeurs prises par des individus permettent de classer automatiquement les champignons dans l'une des deux classes.

	poisonous	edible
cap-shape	c	s
cap-surface	p	
cap-color		r, u, w
odor	c, f, m, p, s, y	a, l
gill-color	b, r	e, o
stalk-root		r
stalk-color-above-ring	b, c, y	e, g, o

stalk-color-below-ring	b, c, y	e, g, o
veil-color	y	n, o
ring-number	p	
ring-type	p	f
spore-print-color	r	b, o, u, y
population		a, n
habitat		w

Malheureusement, aucun attribut ne permet à lui seul de classer les individues dans une classe ou une autre.

L'attribut le plus représentatif est l'odeur car, à l'exception d'une odeur, cet attribut permet de catégoriser les champignons dans la bonne classe.
Cependant, dans ce cas, et encore plus dans les autres, les valeurs qui permettent une séparation nette entre les classes sont celles qui possèdent le moins d'itération.

3. Mise en place en python

Voici le code python utilisé pour pouvoir analyser la base de données grâces aux méthodes vues en RCP209

```
#X contient toute les colonnes de la base de données sauf la 1ere, celle des classes
X=champi.drop('class',axis=1)

#Transformation des valeurs qualitatives en valeurs quantitatives sur X
column_names=list()
for names in X.columns:
    column_names.append(names)
X=pd.get_dummies(data=X, columns=column_names,drop_first=True)

#y ne contient que les classes
y=champi['class']

#Transformation des valeurs qualitatives en valeurs quantitatives sur y
y=pd.get_dummies(y,drop_first=True)
```

Dans un premier temps, les attributs et les classes sont séparés pour bien avoir une base X et y distinctes.

Ensuite, les valeurs qualitatives présentes dans la base de données sont transformées en valeurs quantitatives.

La base de données passe donc de 21 attributs (sans compter la classe) à 116. Chaque attribut devient binaire. Par exemple pour cap-shape, avant la transformation, il pouvait prendre les valeurs b, c, f, k, s et x. Il existe maintenant 6 attributs cap-shape_b, cap-shape_c, cap-shape_f, cap-shape_k, cap-shape_s et cap-shape_x qui sont à 0 si la valeur cap-shape ne correspond pas à leur valeur, et à 1 si la valeur est bonne.

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

5 rows × 22 columns

Base de données avant transformation

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

5 rows × 116 columns

Base de données avant transformation

Ensuite, la base de données est divisée entre base d'apprentissage et base de test.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Pour choisir le taux de division, différents ratios ont été testés sur les différentes méthodes, tant que celui-ci ne dépasse pas 50% pour la base de test.

La valeur par défaut de train_test_split est de 0.25, cependant dans la littérature, la valeur par récurrence de test-size semble être de 0.2. C'est donc celle qui a été retenue.

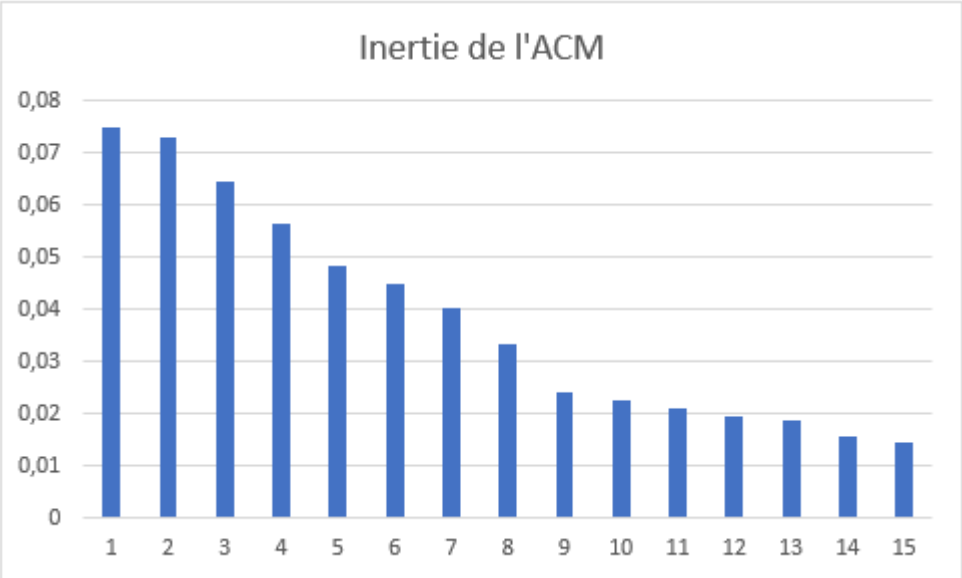
4. ACM & analyse des composantes

Notre base de données étant de nature qualitative nominales, nous devons appliquer un ACM dessus. Vu qu'a l'heure actuel scikit-learn ne propose pas d'implémentation des algorithmes d'AFCB et d'ACM, le paquet prince a été installé et utilisé pour cette partie.

Le code suivant permet d'afficher l'inertie des 15 premiers axes ainsi que le graphique des axes 1 et 2.

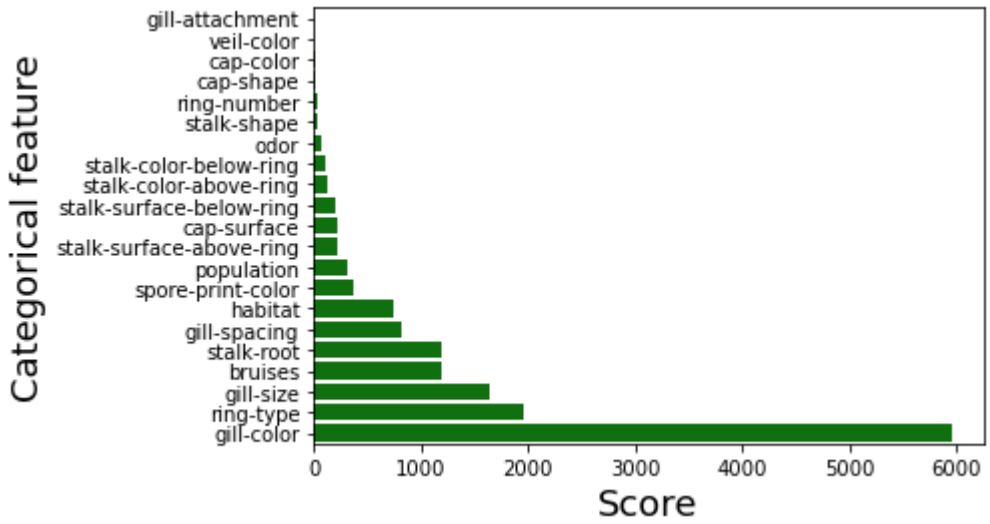
```
import prince
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder
from sklearn.feature_selection import SelectKBest, chi2

#Récupération de la base de données sans les classes
amcchampi=champi.drop('class',axis=1)
amcchampi.columns = column_names

#Application d'une ACM et affichage de la distance entre les variables
mca = prince.MCA(n_components=15, n_iter=3, copy=True, check_input=True, engine='auto', random_state=42)
mca = mca.fit(amcchampi)

# Liste de l'inertie des différents composants
mca.explained_inertia_

ax = mca.plot_coordinates(X=amcchampi, ax=None, figsize=(50, 50), x_component=0, y_component=1, show_row_points=False, row_points_size=10,
                          show_row_labels=False, show_column_points=True, column_points_size=30, show_column_labels=True, legend_n_cols=1)
ax.get_figure().savefig('mca.svg')
```

Nous pouvons voir que les différents axes possèdent peut d'inertie. Pour obtenir seulement 50% d'inertie il faudrait analyser les 11 premières composantes. Vous pouvez trouvez la visualisation des 2 premiers axes en annexe c



```python
#Récupération de la base de données pour l'analyse des features
X1 = db=champi.drop('class',axis=1)
y1 = champi['class']

#Préparation des variables
oe = OrdinalEncoder()
oe.fit(X1)
X_enc = oe.transform(X1)

#Préparation des classes
le = LabelEncoder()
le.fit(y1)
y_enc = le.transform(y1)

#Affichage du score de chaque variable
sf = SelectKBest(chi2, k='all')
sf_fit1 = sf.fit(X_enc, y_enc)
for i in range(len(sf_fit1.scores_)):
    print(' %s: %f' % (X1.columns[i], sf_fit1.scores_[i]))

# Affichage des variables par score
datset1 = pd.DataFrame()
datset1['feature'] = X1.columns[ range(len(sf_fit1.scores_))]
datset1['scores'] = sf_fit1.scores_
datset1 = datset1.sort_values(by='scores', ascending=True)
sns.barplot(datset1['scores'], datset1['feature'], color='green')
sns.set_style('whitegrid')
plt.ylabel('Categorical feature', fontsize=18)
plt.xlabel('Score', fontsize=18)
plt.show()
```

Pour une meilleure compréhension, une analyse des features est lancé avec le code ci-dessus qui nous permet d'obtenir le tableau de score des différentes variables. Nous pouvons voir que la couleur des lames du champignons et la variable qui possède la plus grande influence. En effet la couleur buff (jaune) représente plus de 20% des champignons et appartient toujours a la même classe



5. Analyse grâce à différentes techniques

a. Arbre de données

Voici le code qui a été utilisé pour la recherche en arbre de données :

```
#Partie arbre de décision
from sklearn import tree
from math import *
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV

#Determination des meilleurs parametres pour l'arbre
pgrid = {"max_depth": [1, 2, 3, 4, 5, 6, 7, 8, 9],
      "min_samples_split": [2, 3, 5, 10, 15, 20]}
grid_search = GridSearchCV(tree.DecisionTreeClassifier(), param_grid=pgrid, cv=10)
grid_search.fit(X_train, y_train)
grid_search.best_params_
```

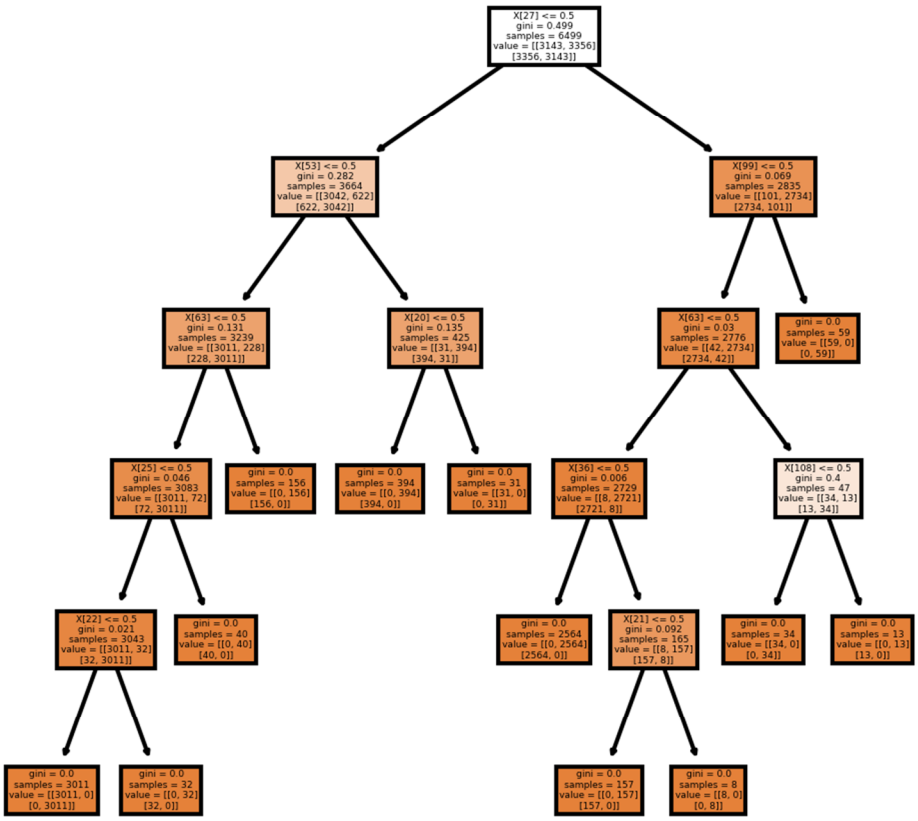
```
{'max_depth': 7, 'min_samples_split': 2}
```

```
clf = tree.DecisionTreeClassifier(min_samples_leaf=7, min_samples_split=2)
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

```
1.0
```

Pour afficher et enregistrer l'arbre, nous utilisons le code suivant :

```
#Affichage de l'arbre
tree.plot_tree(arbre, filled=True)

# On exporte le graphe dans le fichier champignon.dot
with open("arbre.dot", 'w') as f:
    f = tree.export_graphviz(arbre, out_file=f, filled=True)

#Enregistrement de l'arbre pour une meilleur visualisation
plt.savefig('arbre.png')
```



L'utilisation d'arbre a été proposé dans le fichier agaricus-lepiota.NAMES fourni avec la base de données.

"Logical rules for the mushroom data sets.

Logical rules given below seem to be the simplest possible for the

mushroom dataset and therefore should be treated as benchmark results.

Disjunctive rules for poisonous mushrooms, from most general

to most specific:

P_1) odor=NOT(almond.OR.anise.OR.none)

120 poisonous cases missed, 98.52% accuracy

P_2) spore-print-color=green

48 cases missed, 99.41% accuracy

P_3) odor=none.AND.stalk-surface-below-ring=scaly.AND.

(stalk-color-above-ring=NOT.brown)

8 cases missed, 99.90% accuracy

P_4) habitat=leaves.AND.cap-color=white

100% accuracy

Rule P_4) may also be

P_4') population=clustered.AND.cap_color=white

These rule involve 6 attributes (out of 22). Rules for edible

mushrooms are obtained as negation of the rules given above, for

example the rule:

odor=(almond.OR.anise.OR.none).AND.spore-print-color=NOT.green

gives 48 errors, or 99.41% accuracy on the whole dataset.

Several slightly more complex variations on these rules exist,

involving other attributes, such as gill_size, gill_spacing,

stalk_surface_above_ring, but the rules given above are the simplest

we have found.Bagging Classifier"

Cette description des règles logiques de la base de données évoque tout de suite un arbre de décision. En premier lieu un GridSearchCV a été employé pour trouver les meilleurs paramètres pour l'arbre.

Une fois ceux-ci déterminés, ils ont été appliqués à l'arbre.

Celui-ci nous ayant donné un score de 100% de réussite sur la base de test, et étant donné que l'arbre obtenu est équilibré, nous ne sommes pas aller plus loin dans ce type d'analyse comme les forets aléatoires.

b. Bagging, Forêts aléatoires et Boosting

Apres avoir tester les arbres classiques, d'autres méthodes plus avancées de la même famille sont testées : le bagging, les forêts aléatoires, les extra Tree et le Boosting.

Le code suivant est utilisé :

```
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import ExtraTreesClassifier
```

```
#Determination des meilleurs parametres pour le Bagging
pgrid = {"max_samples": [0.1, 0.2, 0.4, 0.6, 0.8],
        "max_features": [0.2, 0.4, 0.6, 0.8, 1]}
grid_search = GridSearchCV(BaggingClassifier(tree.DecisionTreeClassifier()), param_grid=pgrid, cv=5)
grid_search.fit(X_train, y_train)
grid_search.best_params_
```

```
#Test avec le Bagging
bagging = BaggingClassifier(tree.DecisionTreeClassifier(),
                            max_samples=grid_search.best_estimator_.max_samples,
                            max_features=grid_search.best_estimator_.max_features,
                            n_estimators=200)
bagging.fit(X_train, y_train)
```

```
#Test avec les Forêts aléatoires
random = RandomForestClassifier(n_estimators=200)
random.fit(X_train, y_train)
```

```
extra = ExtraTreesClassifier(n_estimators=200)
extra.fit(X_train, y_train)
```

```
#Test avec le Boosting
boost = AdaBoostClassifier(base_estimator=tree.DecisionTreeClassifier(max_depth=5),
                           n_estimators=200, learning_rate=2)
boost.fit(X_train, y_train)
```

```
print("Bagging (200 arbres) : " + str(bagging.score(X_test,y_test)))
print("Forêt aléatoire (200 arbres) : " + str(random.score(X_test,y_test)))
print("Extra Trees (200 arbres) : " + str(extra.score(X_test,y_test)))
print("AdaBoost (200 arbres) : " + str(boost.score(X_test,y_test)))
```

Comme avec les arbres, les meilleurs paramètres sont déterminés grâce à un GridSearchCV.

Au final les résultats sont similaires a celui de la classification en arbre classique avec une précision de 100% pour chaque méthode.

c. SVM

Voici le code qui a été utilisé pour la recherche en SVM

```
from sklearn import svm, datasets

#Determination des meilleurs parametres pour le SVM linéaire
pgrid = {"C": [1, 8, 10, 15]}
grid_search = GridSearchCV(svm.LinearSVC(), param_grid=pgrid, cv=5)
grid_search.fit(X_train, y_train)
print("Meilleur C: " + str(grid_search.best_estimator_.C))
```

```
lin_svc = svm.LinearSVC(C=grid_search.best_estimator_.C)
lin_svc.fit(X_train, y_train)
lin_svc.score(X_test, y_test)
```

```
#Determination des meilleurs parametres pour le SVM
pgrid = {"C": [1, 8, 10, 15],
        "gamma": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]}
grid_search = GridSearchCV(svm.SVC(), param_grid=pgrid, cv=5)
grid_search.fit(X_train, y_train)
print("Meilleur C: " + str(grid_search.best_estimator_.C))
print("Meilleur gamma: " + str(grid_search.best_estimator_.gamma))
```

```
clf = svm.SVC(C=grid_search.best_estimator_.C, kernel='rbf', gamma=grid_search.best_estimator_.gamma)
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

Comme pour les méthodes précédentes, une GridSearchCV est utilisé pour déterminer les paramètres optimaux. Grace a cela, les deux méthodes, linéaire ou non, on un score de 1.

Cependant la recherche de paramètre pour le SVM non linéaire prend beaucoup plus de temps que toutes les autres méthodes de ce projet. Il n'est donc pas conseillé de l'utiliser tel quel.

d. MLP

Voici le code qui a été utilisé pour la recherche en MLP :

```
#Partie Réseau de neurones profond
import tensorflow
from tensorflow import keras
from keras import layers, Sequential
from keras.layers import Dense
```

```
# Initialisation du reseau de neurones profond
analyse_champi = Sequential()

#Ajout des couches
analyse_champi.add(Dense(3, activation = 'relu',  input_dim=116, name='couche_1'))
analyse_champi.add(Dense(3, activation='relu', name='couche_2'))
analyse_champi.add(Dense(2 ,activation='softmax', name='couche_final'))

#Compilation
analyse_champi.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

#Fit (possibilité de modification du verbose pour afficher ou non les différentes epochs
historique = analyse_champi.fit(X_train, y_train, batch_size = 80, epochs = 25,validation_data=(X_test,y_test),verbose=1)
print('---------------------------------')

#Affichage du score et d'un graphique montrant l'evolution des différentes valeurs
pd.DataFrame(historique.history).plot()
scores = analyse_champi.evaluate(X_test, y_test, verbose=0)
print("%s: %.2f%%" % (analyse_champi.metrics_names[0], scores[0]*100))
print("%s: %.2f%%" % (analyse_champi.metrics_names[1], scores[1]*100))
```

Pour ce réseau de neurones plusieurs configurations ont été testées, en changeant le nombre de couches, le nombre de neurones, les activations, la loss et l'optimizer. Les tests ont été faits plusieurs fois pour obtenir une valeur moyenne de la loss, de l'accuracy et du nombre d'epochs avant que l'analyse ne se stabilise.

Deux optimizer ont été testés : la descente de gradient stochastique et adam. Le reste du réseau était composé comme dans l'image suivante :

```
# Initialisation du reseau de neurones profond
analyse_champi = Sequential()

#Ajout des couches
analyse_champi.add(Dense(3, activation = 'relu',  input_dim=116, name='couche_1'))
analyse_champi.add(Dense(3, activation='relu', name='couche_2'))
analyse_champi.add(Dense(2 ,activation='softmax', name='couche_final'))

#Compilation
analyse_champi.compile(loss='binary_crossentropy',optimizer=[A tester],metrics=['accuracy'])

#Fit (possibilité de modification du verbose pour afficher ou non les différentes epochs
historique = analyse_champi.fit(X_train, y_train, batch_size = 80, epochs = 25,validation_data=(X_test,y_test),verbose=1)
print('--------------------------------')

#Affichage du score et d'un graphique montrant l'evolution des différentes valeurs
pd.DataFrame(historique.history).plot()
scores = analyse_champi.evaluate(X_test, y_test, verbose=0)
print("%s: %.2f%%" % (analyse_champi.metrics_names[0], scores[0]*100))
print("%s: %.2f%%" % (analyse_champi.metrics_names[1], scores[1]*100))
```

Les deux arrivent à 100% d'accuracy en moins de 10 epochs mais adam met environ 5 epochs de plus pour arriver à ce résultat. De plus sa loss est très légèrement supérieure (0.07 contre 0.01).

La descente de gradient stochastique de gradient a donc été choisie.

Deux loss ont été testés : categorical_crossentropy et binary_crossentropy.
Le paramétrage suivant a été utilisé :

```
# Initialisation du reseau de neurones profond
analyse_champi = Sequential()

#Ajout des couches
analyse_champi.add(Dense(3, activation = 'relu',  input_dim=116, name='couche_1'))
analyse_champi.add(Dense(3, activation='relu', name='couche_2'))
analyse_champi.add(Dense(2 ,activation='softmax', name='couche_final'))

#Compilation
analyse_champi.compile(loss=[A tester],optimizer='adam',metrics=['accuracy'])

#Fit (possibilité de modification du verbose pour afficher ou non les différentes epochs
historique = analyse_champi.fit(X_train, y_train, batch_size = 80, epochs = 25,validation_data=(X_test,y_test),verbose=1)
print('--------------------------------')

#Affichage du score et d'un graphique montrant l'evolution des différentes valeurs
pd.DataFrame(historique.history).plot()
scores = analyse_champi.evaluate(X_test, y_test, verbose=0)
print("%s: %.2f%%" % (analyse_champi.metrics_names[0], scores[0]*100))
print("%s: %.2f%%" % (analyse_champi.metrics_names[1], scores[1]*100))
```

La méthode de loss categorical_crossentropy présente de moins bon résultat que la binary_crossentropy.

Le nombre de couches et de neurones par couche changent grandement la vitesse de convergence du réseau. Celui-ci fonctionne avec 2 couches et 3 neurones dans la première couche, cependant il converge beaucoup plus vite avec 3 couches et 150 neurones. Il s'agira donc de déterminer les besoins d'une analyse et savoir si les performances doivent être prioritisés par rapport à son coût machine.

Dans notre cas, le choix d'un réseau peu gourmand en ressources a été fait pour soulager le serveur du CNAM.

Pour les méthodes d'activation, nous avons testé relu, sigmoid et softmax mais aucune différence majeure n'a été observée, l'activation relu et softmax pour la couche finale a été choisie au vu des différents exemples de la littérature et du cours.

6. Conclusion

Deux méthodes nous permettent donc de trouver une méthode distinguant les champignons comestibles des toxiques.

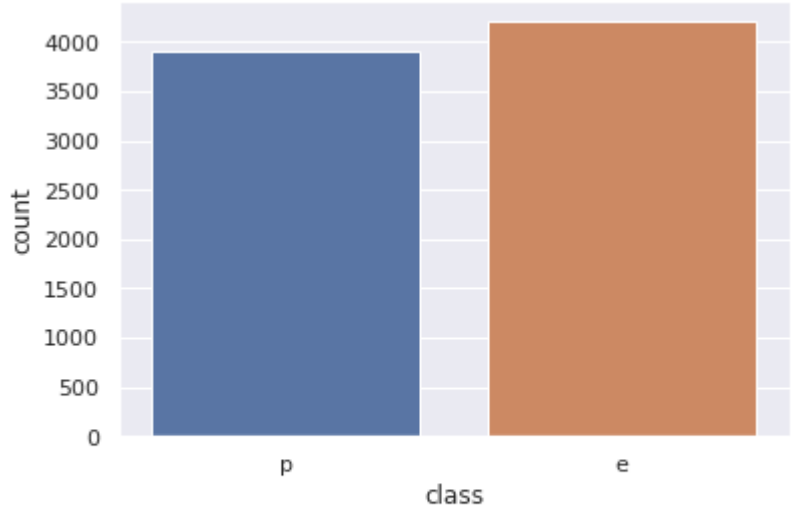
La méthode des arbres de données peut se montrer plus utile car elle nous donne une méthode pour reconnaître manuellement les champignons.

Cependant, il est arrivé de ne pas avoir une accuracy de 100% mais de 98% pour cette méthode, suivant les découpages entre les bases de test et d'entrainement. Il semble donc plus prudent d'utiliser les autres méthodes qui ont toujours affichés une réussite totale.

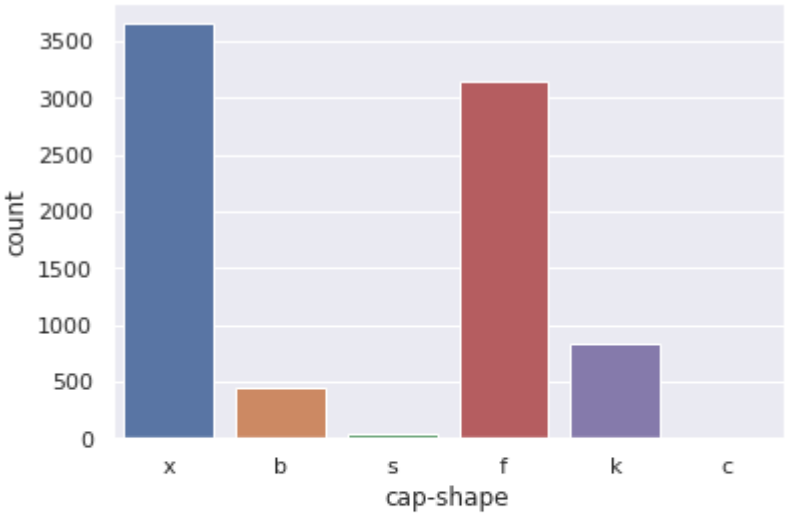
Cependant, quelle que soit la méthode employée, nous pouvons affirmer que le Audubon Society Field Guide to North American Mushrooms est dans l'erreur : il existe une méthode pour déterminer la comestibilité d'un champignon grâce à un arbre.

7. Annexes

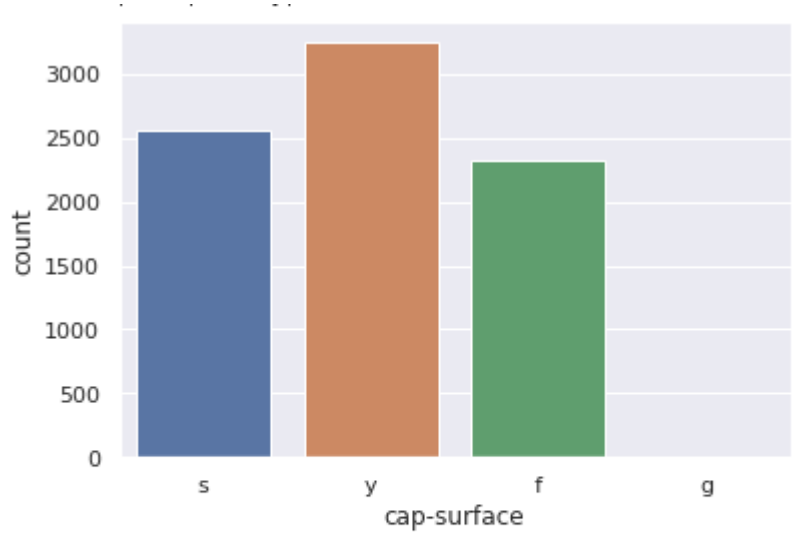
a. Analyse univariée



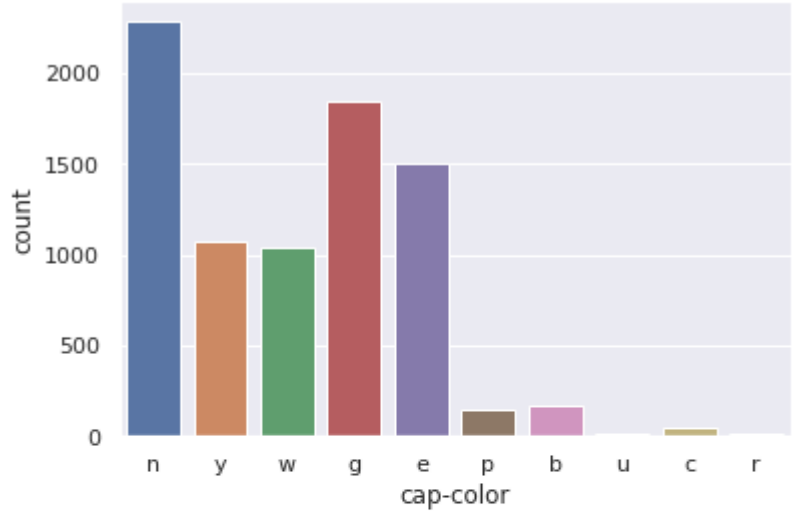

```
% des differentes valeurs:
e    51.8
p    48.2
Name: class, dtype: float64
```



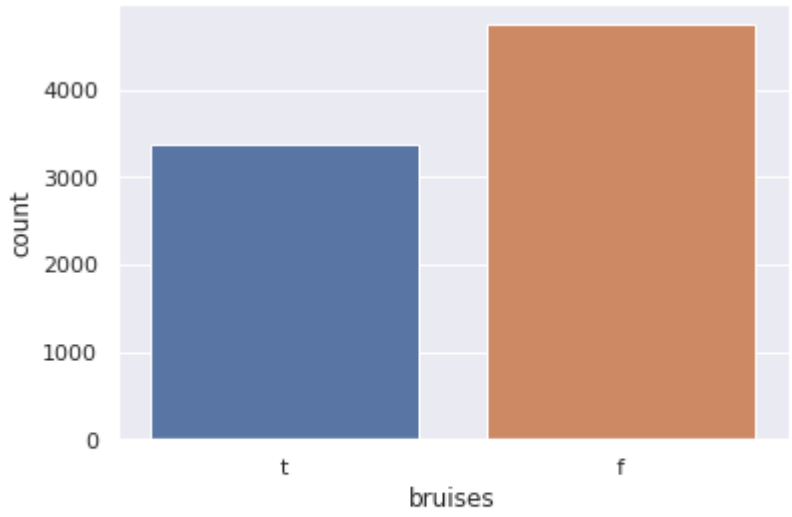

```
% des differentes valeurs:
x    45.00
f    38.80
k    10.19
b     5.56
s     0.39
c     0.05
Name: cap-shape, dtype: float64
```



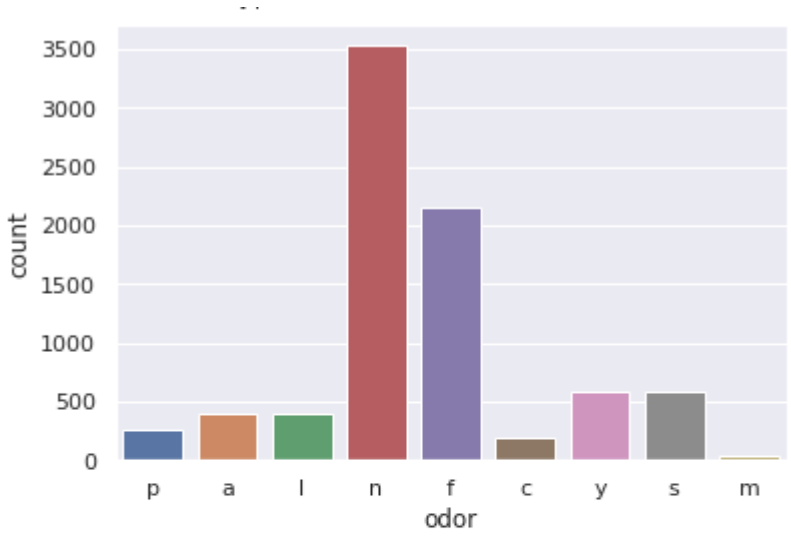

```
% des differentes valeurs:
y    39.93
s    31.46
f    28.56
g     0.05
Name: cap-surface, dtype: float64
```



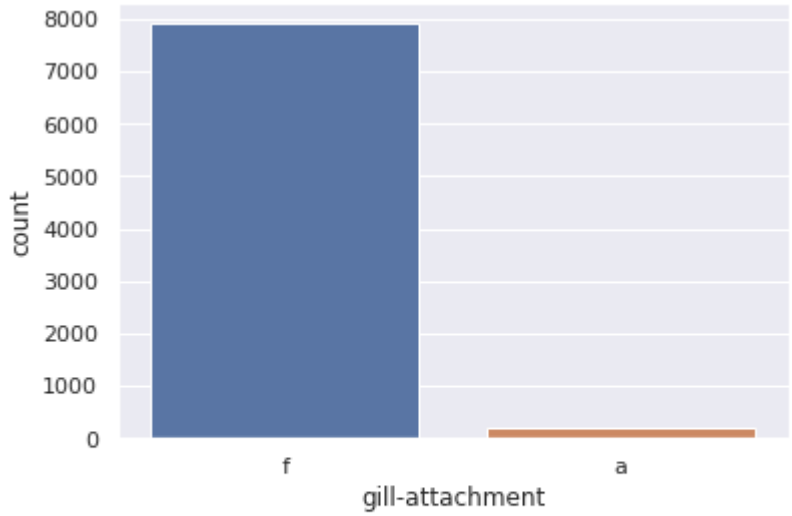

```
% des differentes valeurs:
n    28.11
g    22.65
e    18.46
y    13.20
w    12.80
b     2.07
p     1.77
c     0.54
u     0.20
r     0.20
Name: cap-color, dtype: float64
```



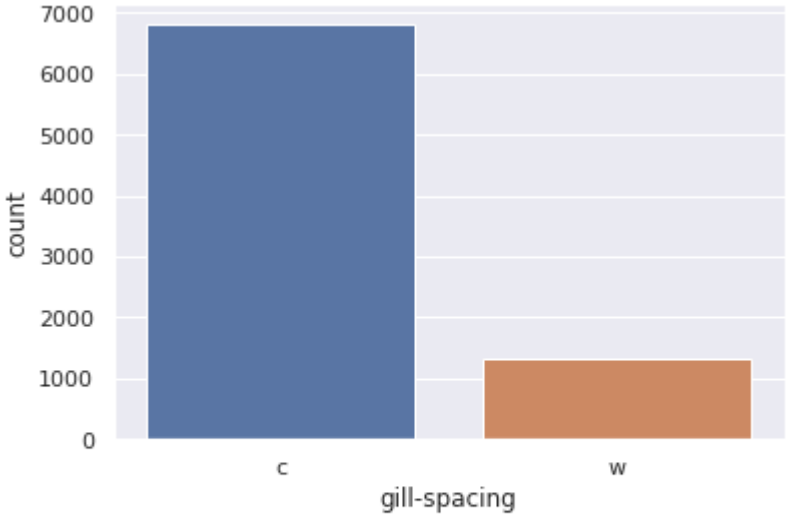

```
% des differentes valeurs:
f    58.44
t    41.56
Name: bruises, dtype: float64
```



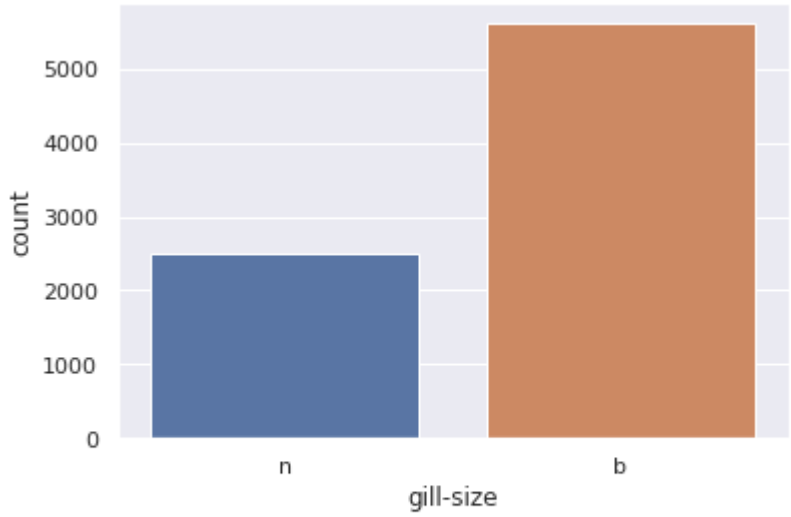

```
% des differentes valeurs:
n    43.43
f    26.59
y     7.09
s     7.09
a     4.92
l     4.92
p     3.15
c     2.36
m     0.44
Name: odor, dtype: float64
```



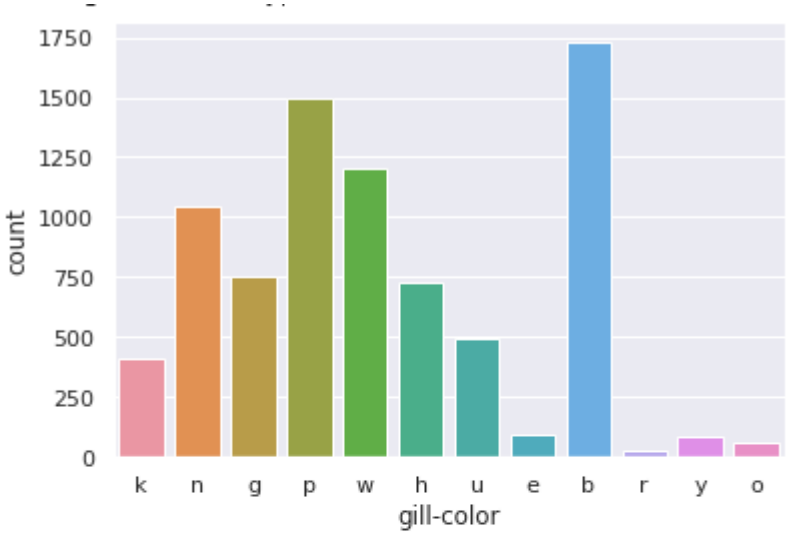

```
% des differentes valeurs:
f    97.42
a     2.58
Name: gill-attachment, dtype: float64
```



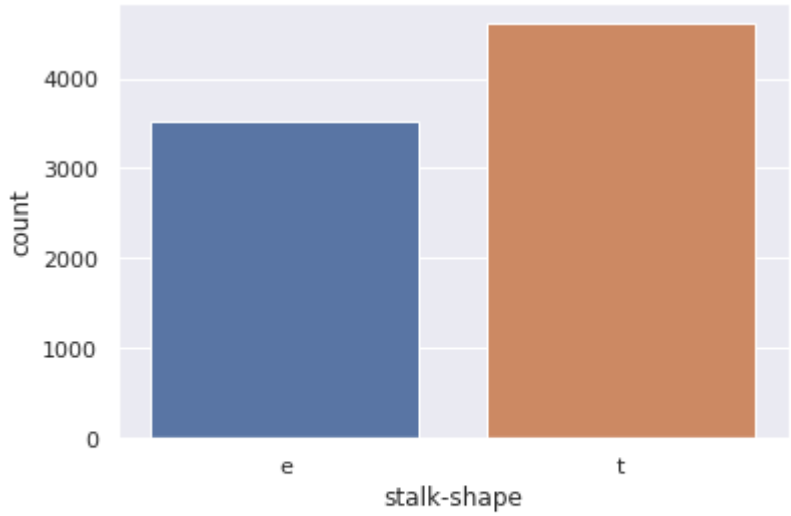

```
% des differentes valeurs:
c    83.85
w    16.15
Name: gill-spacing, dtype: float64
```



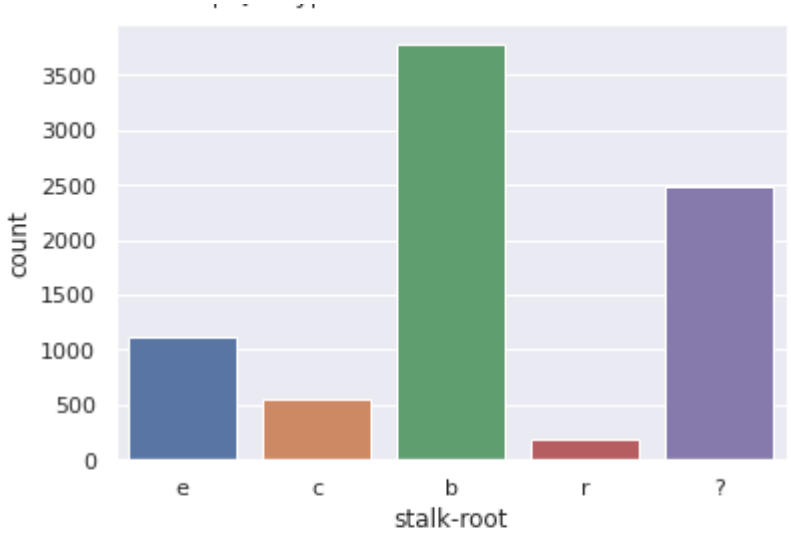

```
% des differentes valeurs:
b    69.08
n    30.92
Name: gill-size, dtype: float64
```



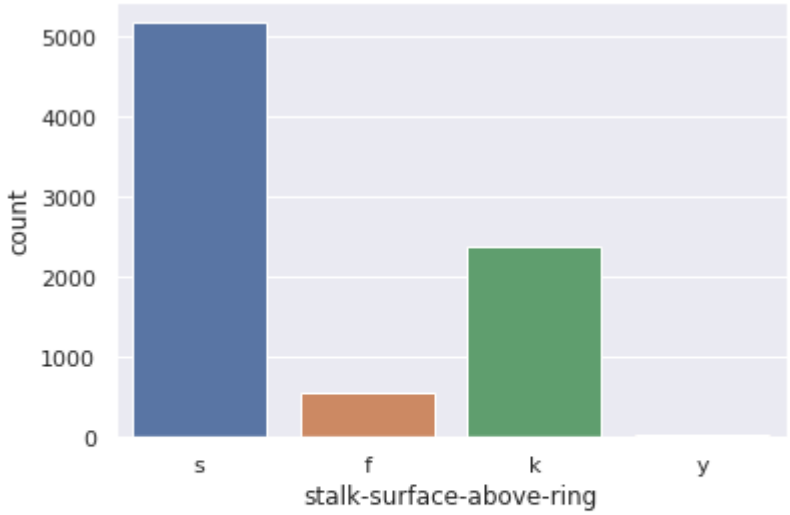

```
% des differentes valeurs:
b    21.27
p    18.37
w    14.80
n    12.90
g     9.26
h     9.01
u     6.06
k     5.02
e     1.18
y     1.06
o     0.79
r     0.30
Name: gill-color, dtype: float64
```



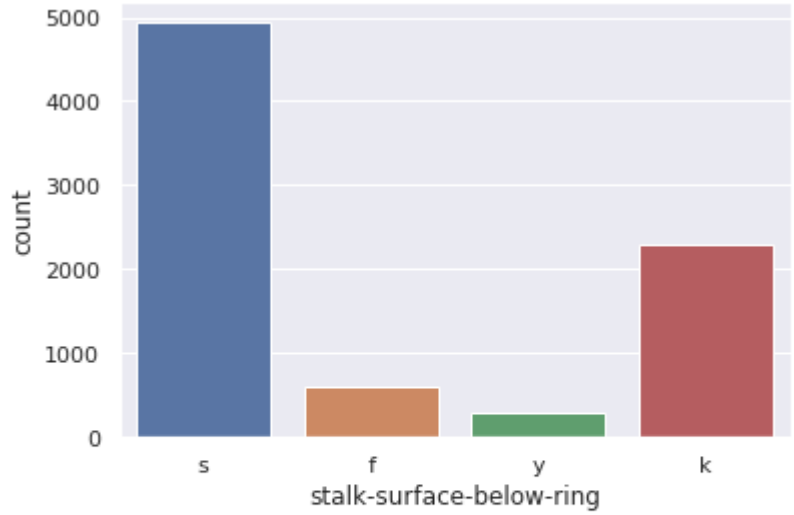

```
% des differentes valeurs:
t    56.72
e    43.28
Name: stalk-shape, dtype: float64
```



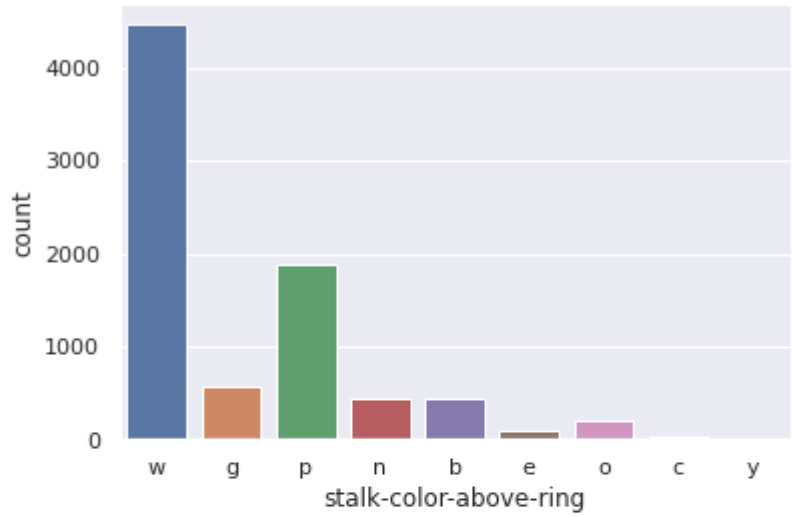

```
% des differentes valeurs:
b    46.48
?    30.53
e    13.79
c     6.84
r     2.36
Name: stalk-root, dtype: float64
```



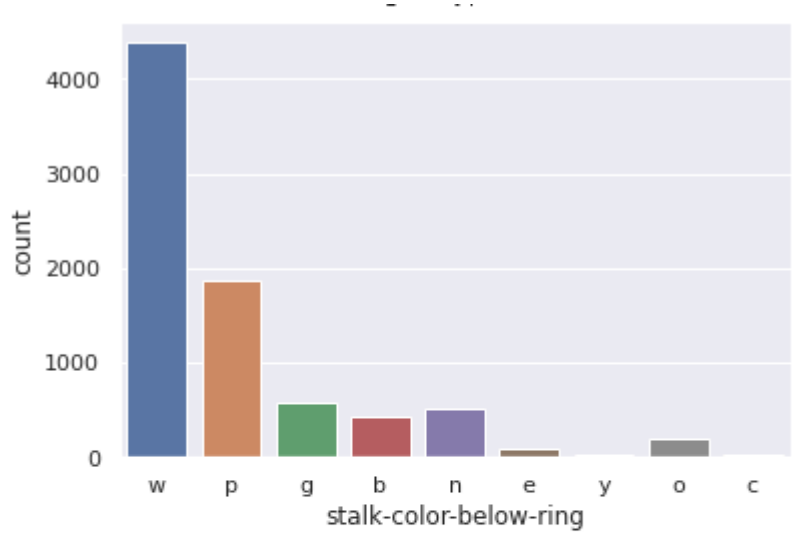

```
% des differentes valeurs:
s    63.71
k    29.20
f     6.79
y     0.30
Name: stalk-surface-above-ring, dtype: float64
```



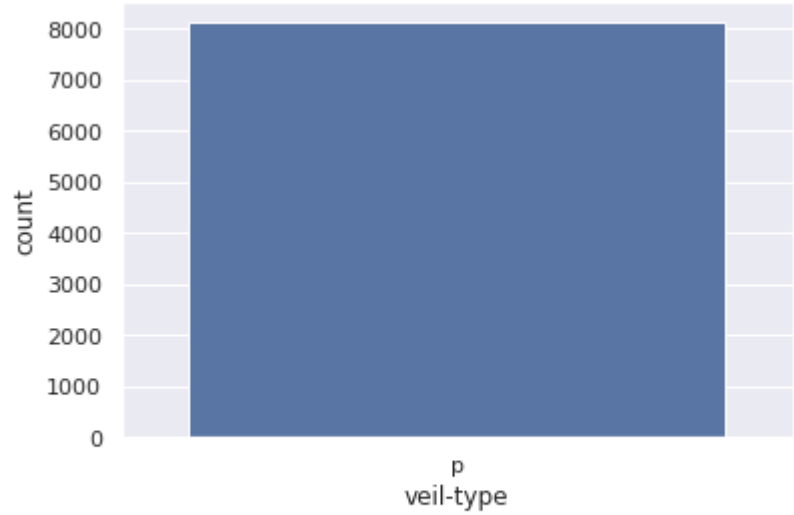

```
% des differentes valeurs:
s    60.76
k    28.36
f     7.39
y     3.50
Name: stalk-surface-below-ring, dtype: float64
```



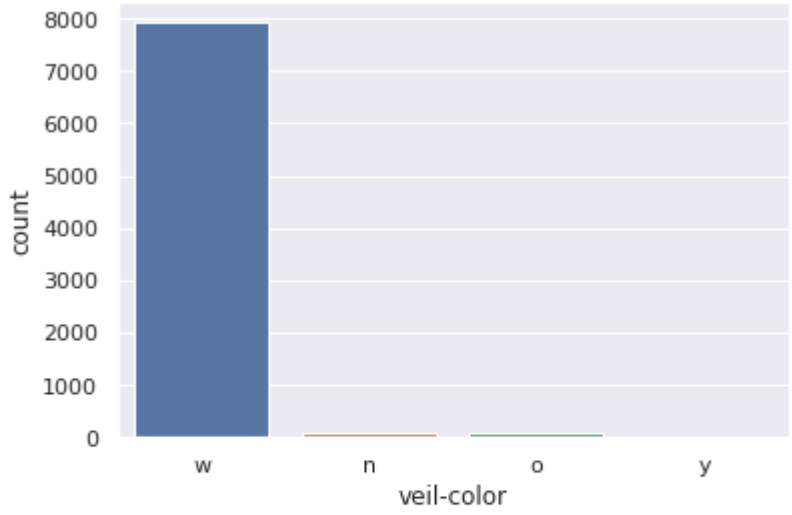

```
% des differentes valeurs:
w    54.95
p    23.04
g     7.09
n     5.51
b     5.32
o     2.36
e     1.18
c     0.44
y     0.10
Name: stalk-color-above-ring, dtype: float64
```



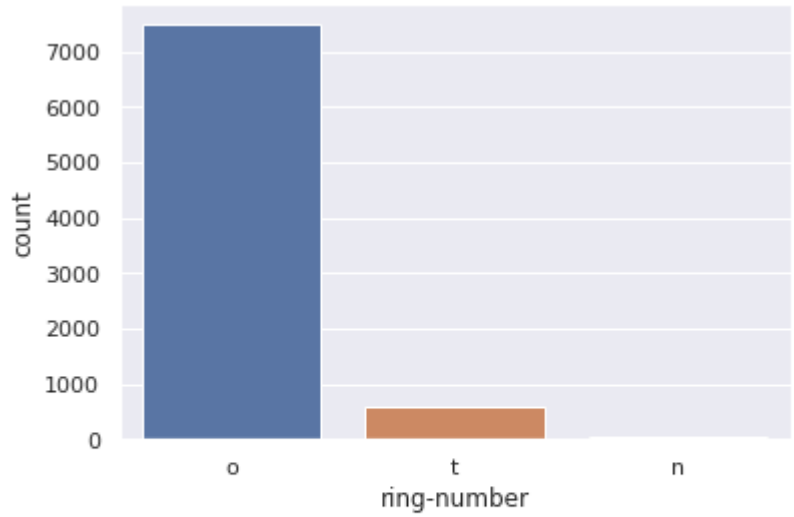

```
% des differentes valeurs:
w    53.96
p    23.04
g     7.09
n     6.30
b     5.32
o     2.36
e     1.18
c     0.44
y     0.30
Name: stalk-color-below-ring, dtype: float64
```



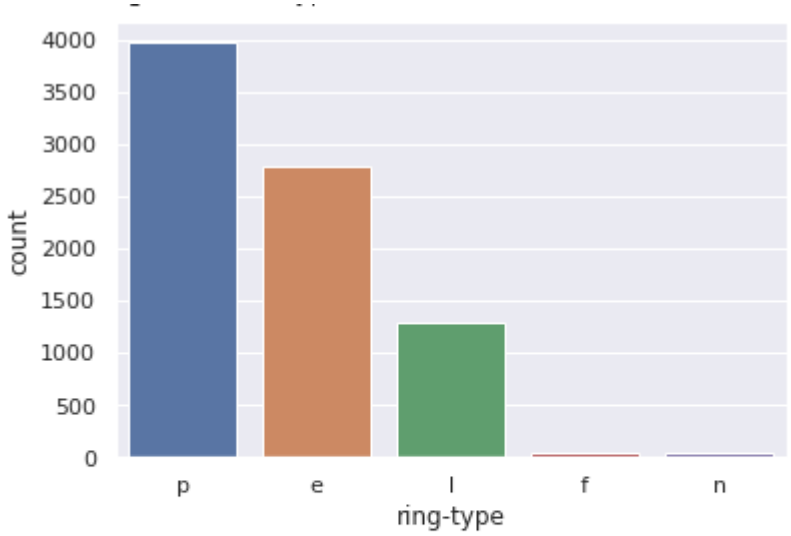

```
% des differentes valeurs:
p    100.0
Name: veil-type, dtype: float64
```



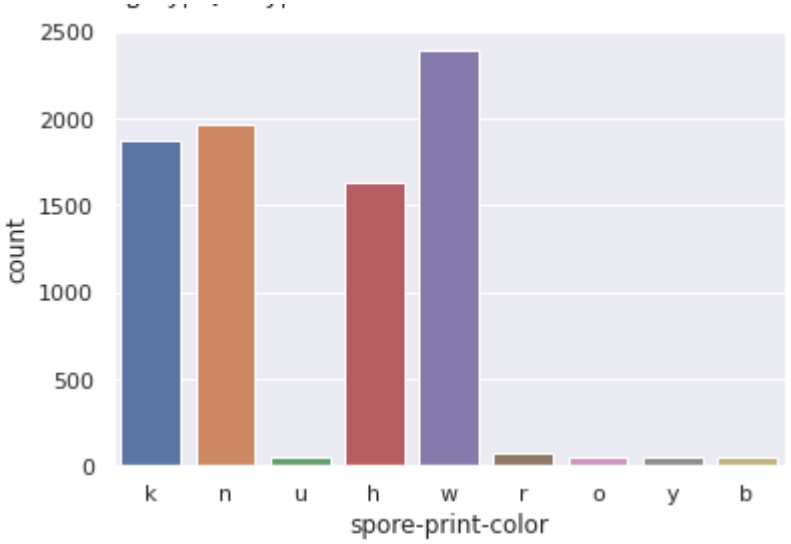

```
% des differentes valeurs:
w    97.54
n     1.18
o     1.18
y     0.10
Name: veil-color, dtype: float64
```



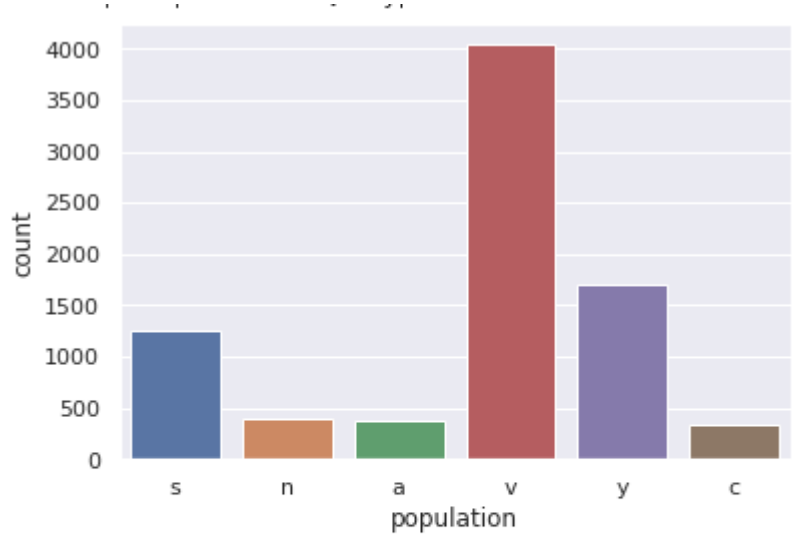

```
% des differentes valeurs:
o    92.17
t     7.39
n     0.44
Name: ring-number, dtype: float64
```



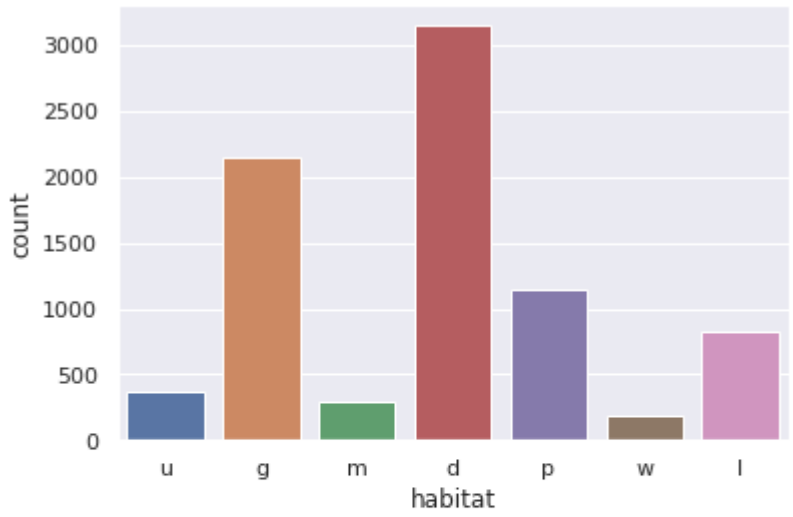

```
% des differentes valeurs:
p    48.84
e    34.17
l    15.95
f     0.59
n     0.44
Name: ring-type, dtype: float64
```




```
% des differentes valeurs:
w    29.39
n    24.22
k    23.04
h    20.09
r     0.89
u     0.59
o     0.59
y     0.59
b     0.59
Name: spore-print-color, dtype: float64
```

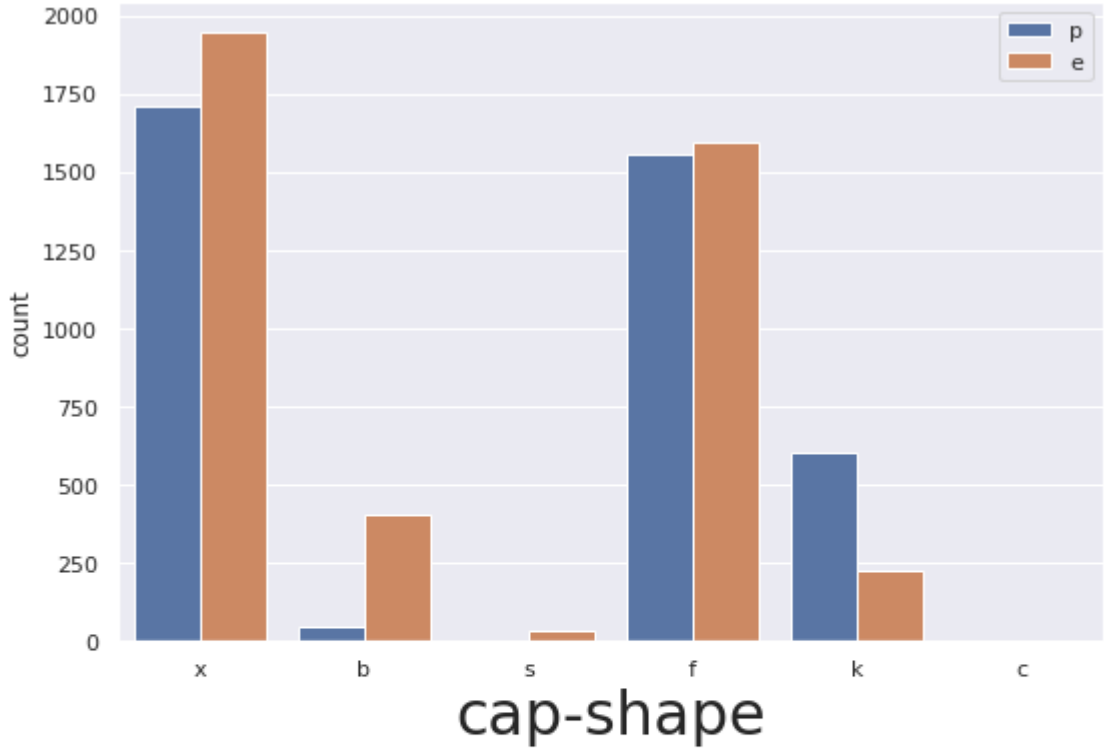



```
% des differentes valeurs:
v    49.73
y    21.07
s    15.36
n     4.92
a     4.73
c     4.19
Name: population, dtype: float64
```

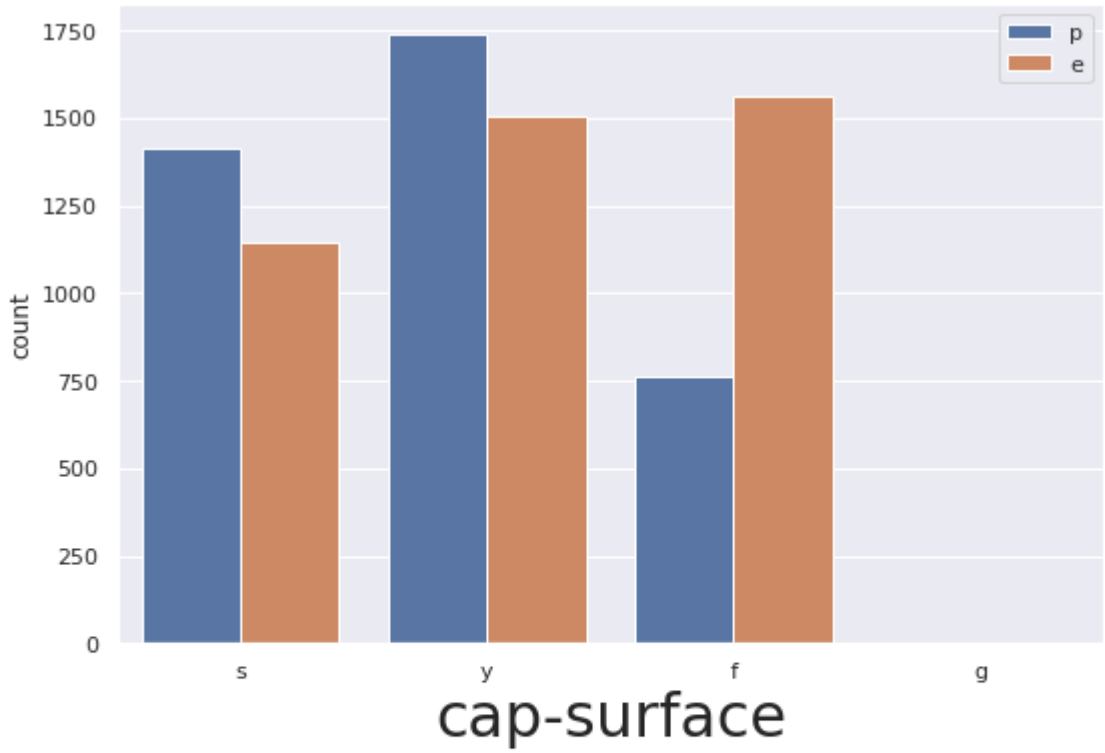



```
% des differentes valeurs:
d    38.75
g    26.44
p    14.08
l    10.24
u     4.53
m     3.59
w     2.36
Name: habitat, dtype: float64
```

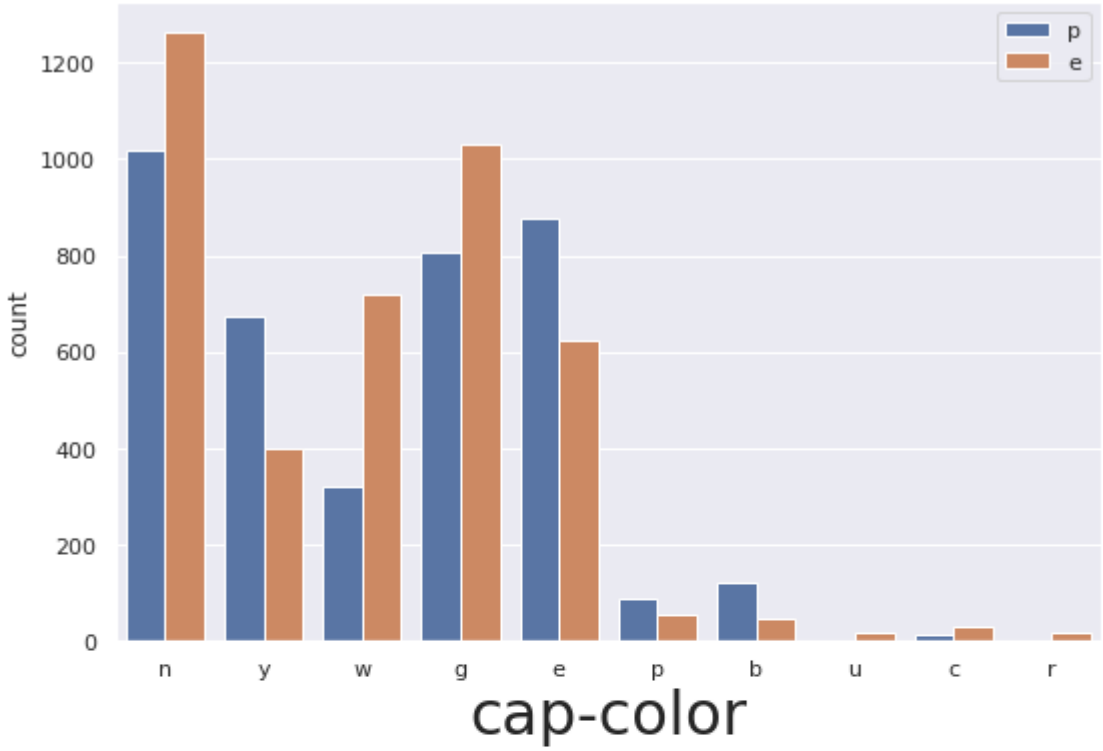
b. Analyse bivariée



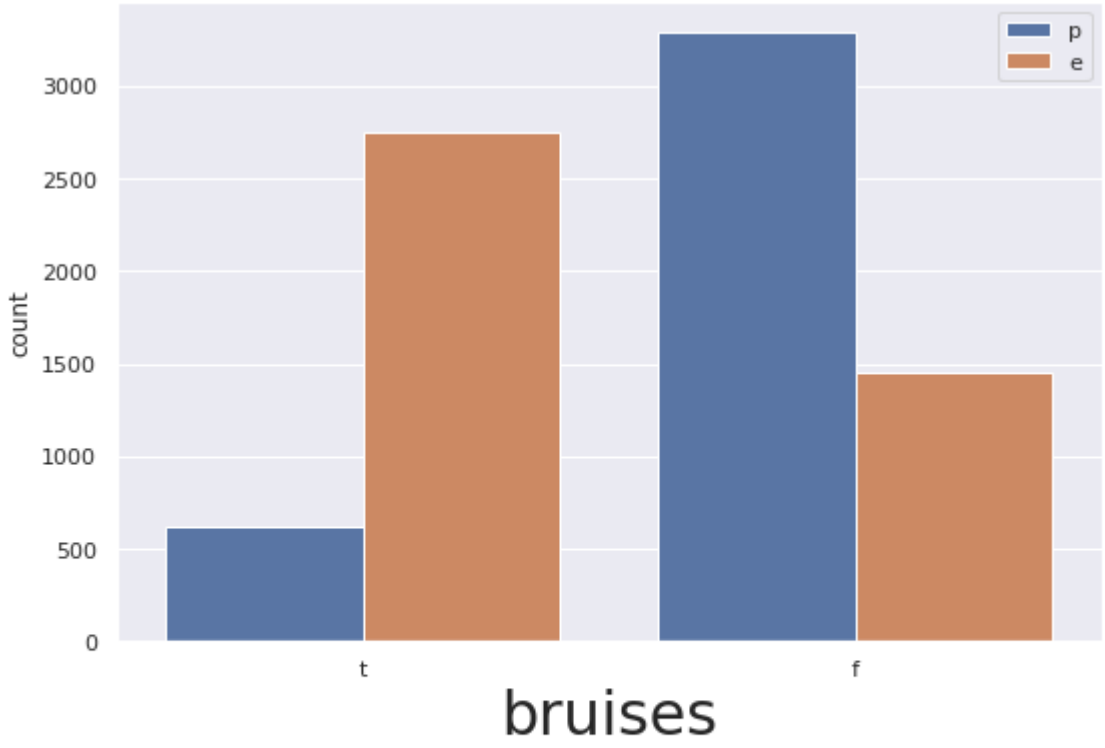
		cap-shape
cap-shape	class	
b	e	404
	p	48
c	p	4
f	e	1596
	p	1556
k	e	228
	p	600
s	e	32
x	e	1948
	p	1708



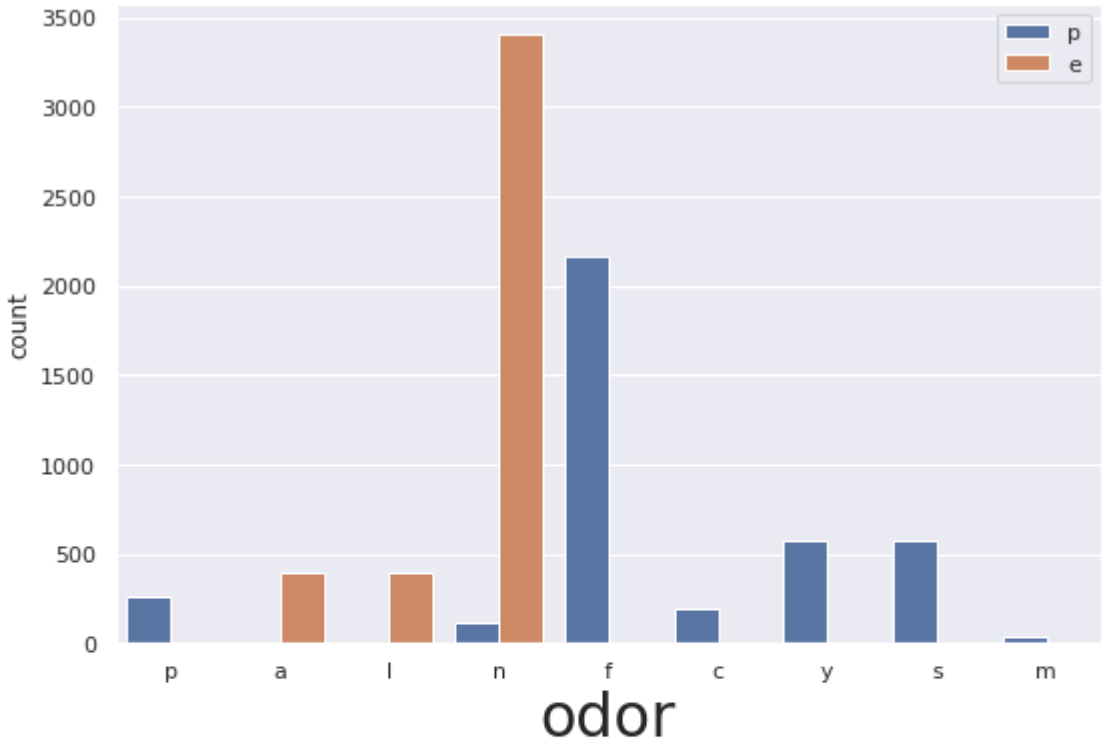
		cap-surface
cap-surface	class	
f	e	1560
	p	760
g	p	4
s	e	1144
	p	1412
y	e	1504
	p	1740



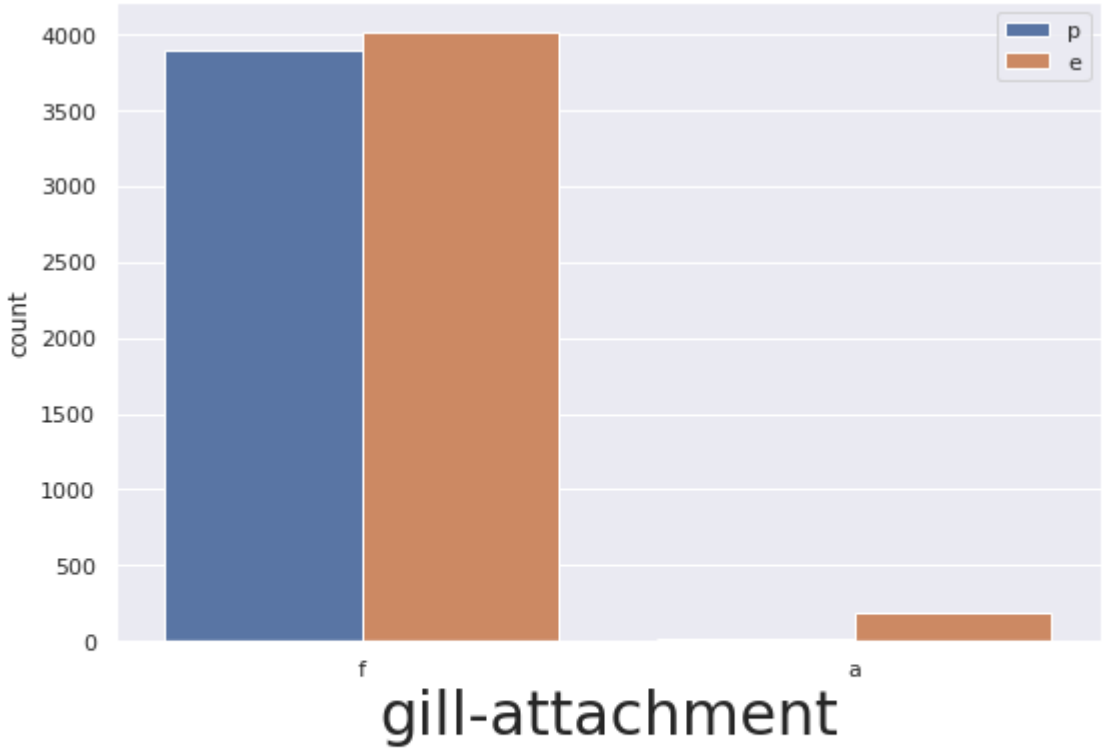
		cap-color
cap-color	class	
b	e	48
	p	120
c	e	32
	p	12
e	e	624
	p	876
g	e	1032
	p	808
n	e	1264
	p	1020
p	e	56
	p	88
r	e	16
u	e	16
w	e	720
	p	320
y	e	400
	p	672



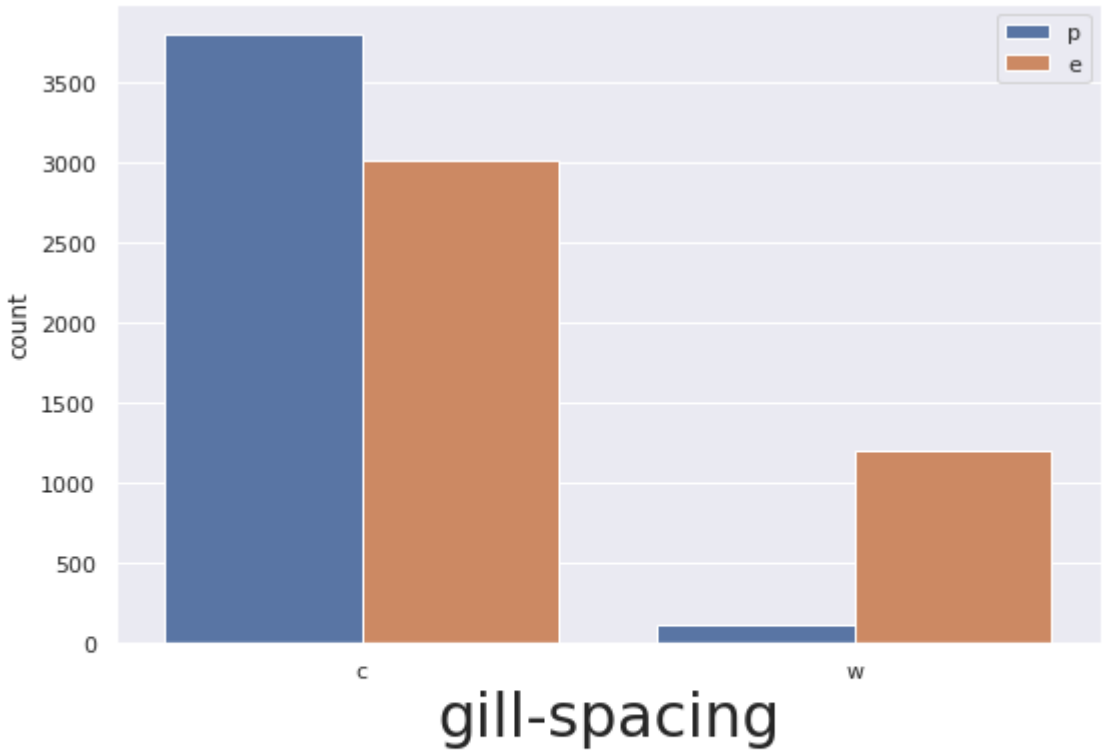
		bruises
bruises	class	
f	e	1456
	p	3292
t	e	2752
	p	624



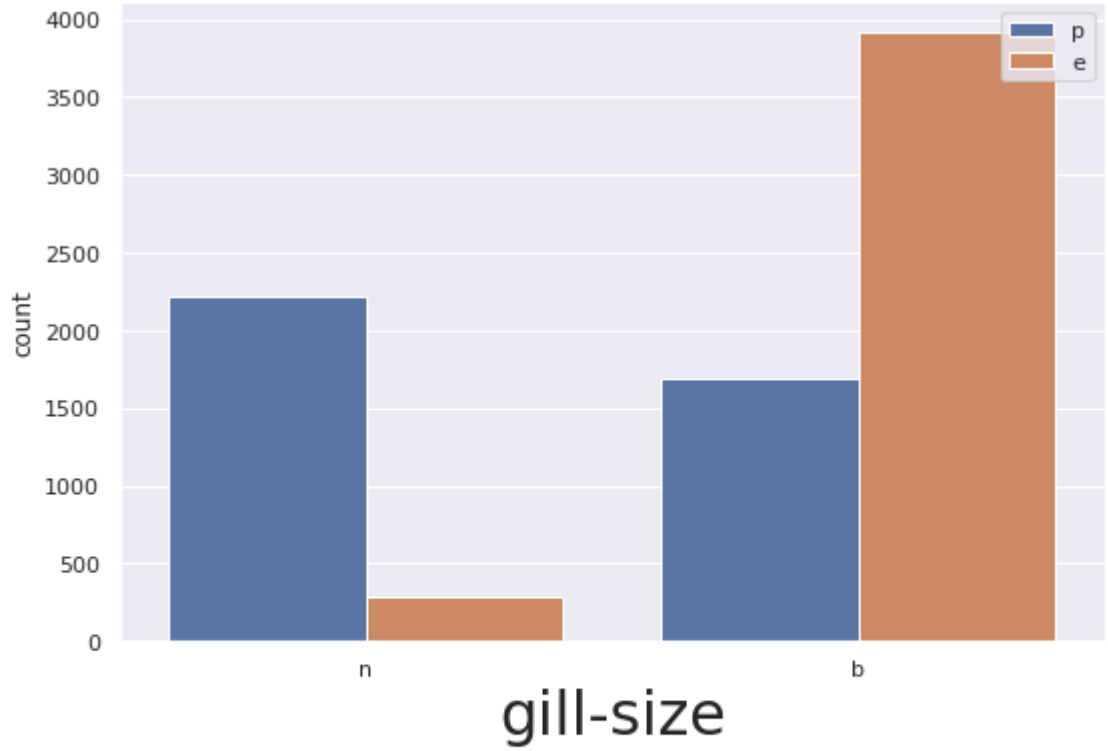
odor	class	odor
a	e	400
c	p	192
f	p	2160
l	e	400
m	p	36
n	e	3408
	p	120
p	p	256
s	p	576
y	p	576



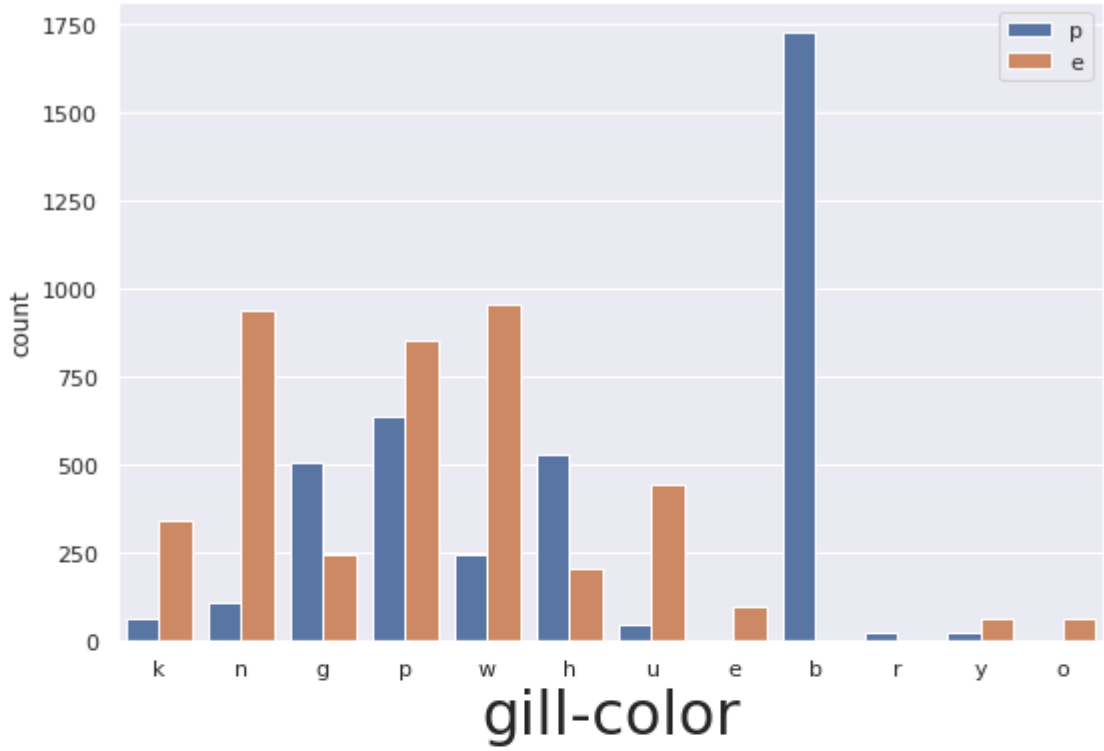
		gill-attachment
gill-attachment	class	
a	e	192
	p	18
f	e	4016
	p	3898



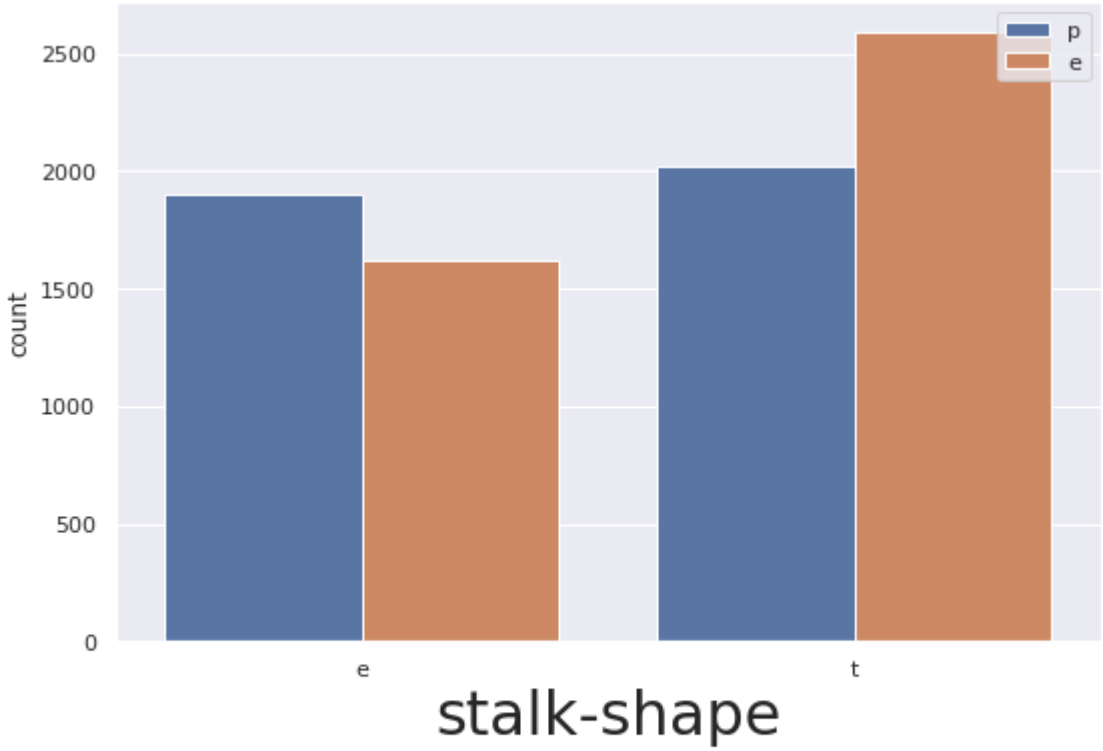
		gill-spacing
gill-spacing	class	
c	e	3008
	p	3804
w	e	1200
	p	112



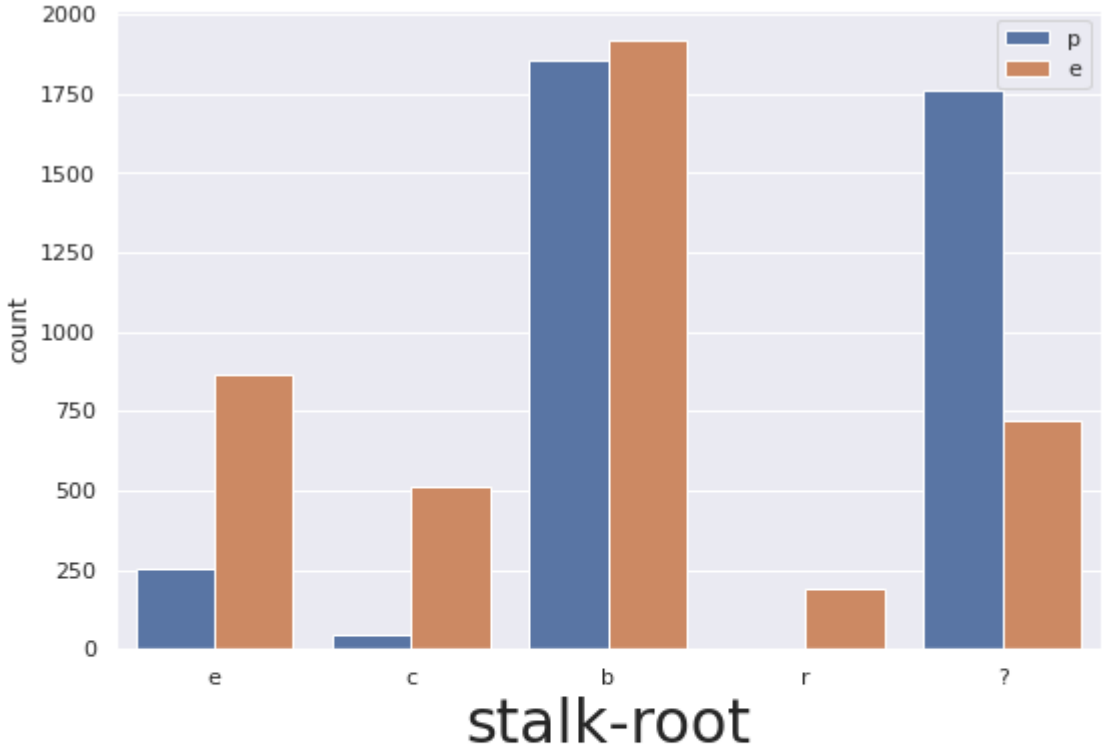
		gill-size
gill-size	class	
b	e	3920
	p	1692
n	e	288
	p	2224



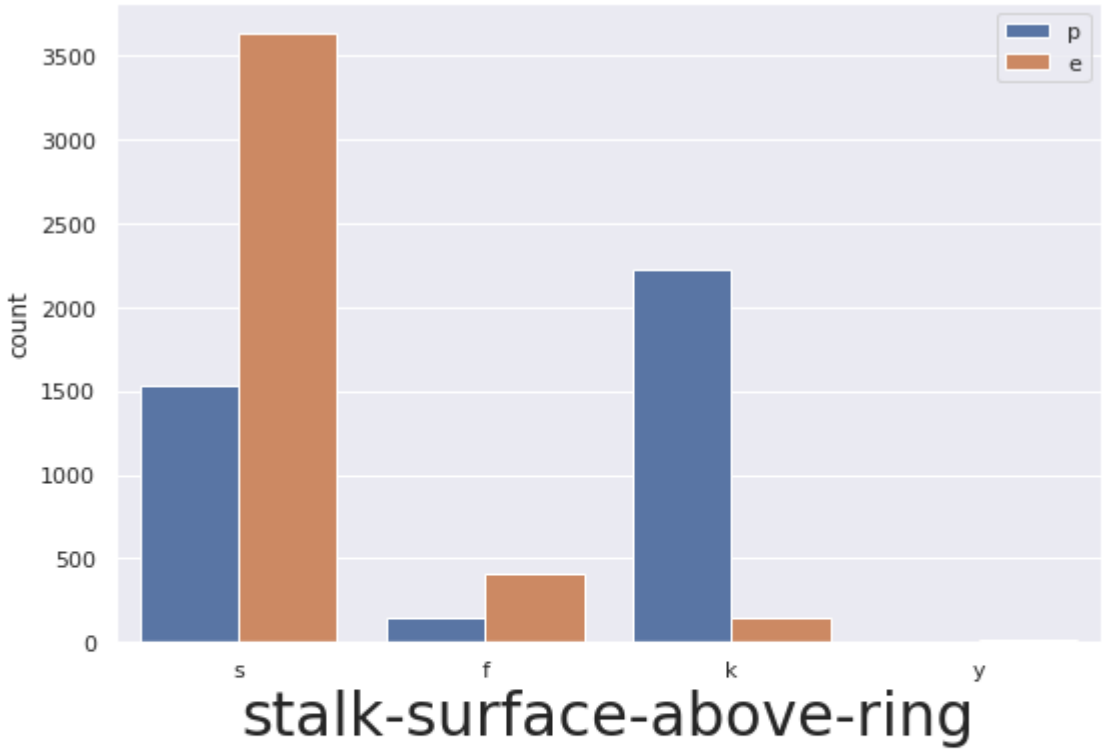
		gill-color
gill-color	class	
b	p	1728
e	e	96
g	e	248
	p	504
h	e	204
	p	528
k	e	344
	p	64
n	e	936
	p	112
o	e	64
p	e	852
	p	640
r	p	24
u	e	444
	p	48
w	e	956
	p	246
y	e	64
	p	22



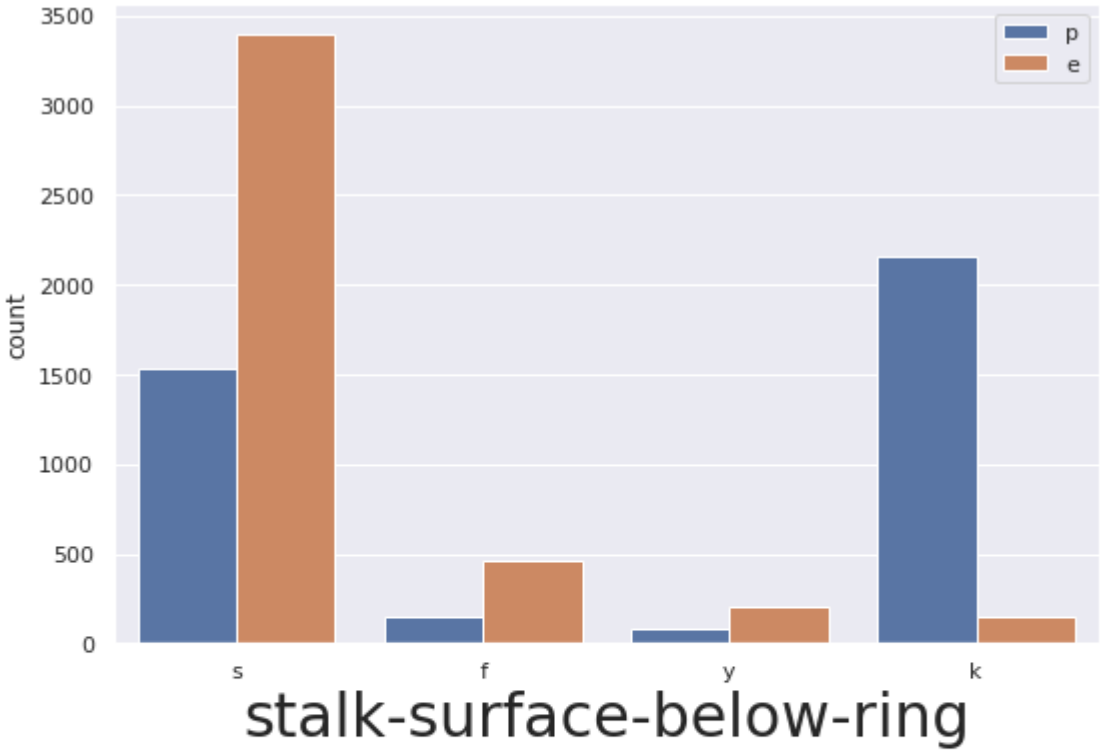
		stalk-shape
stalk-shape	class	
e	e	1616
	p	1900
t	e	2592
	p	2016



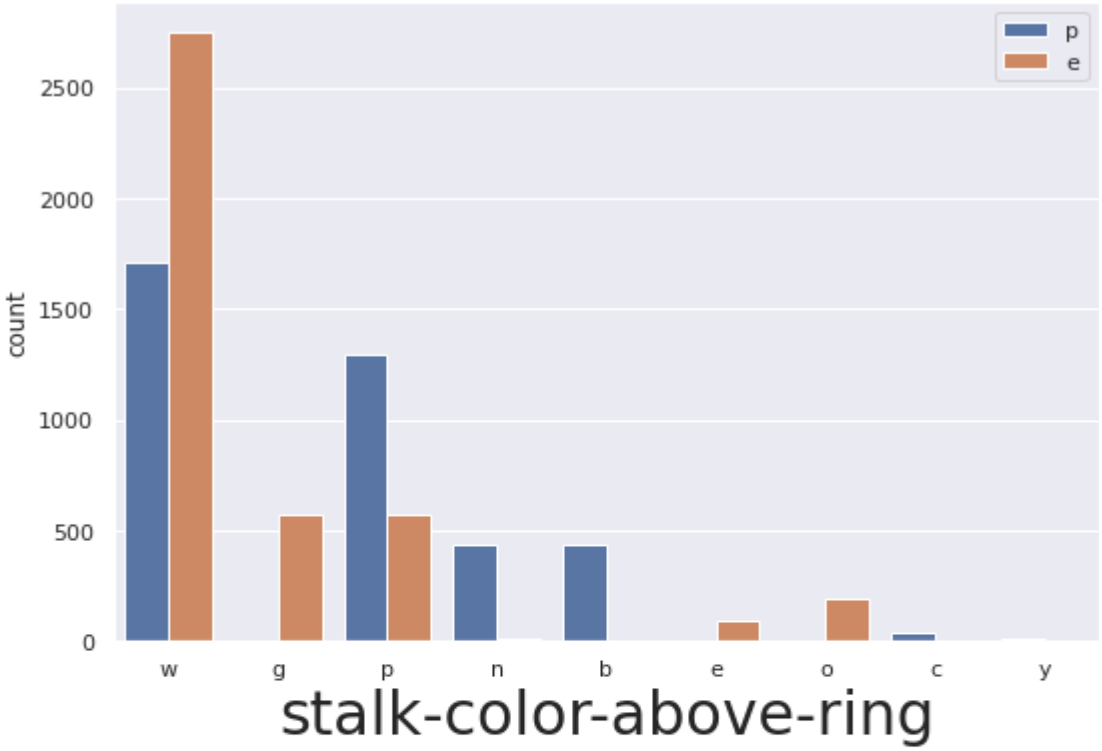
		stalk-root
stalk-root	class	
?	e	720
	p	1760
b	e	1920
	p	1856
c	e	512
	p	44
e	e	864
	p	256
r	e	192



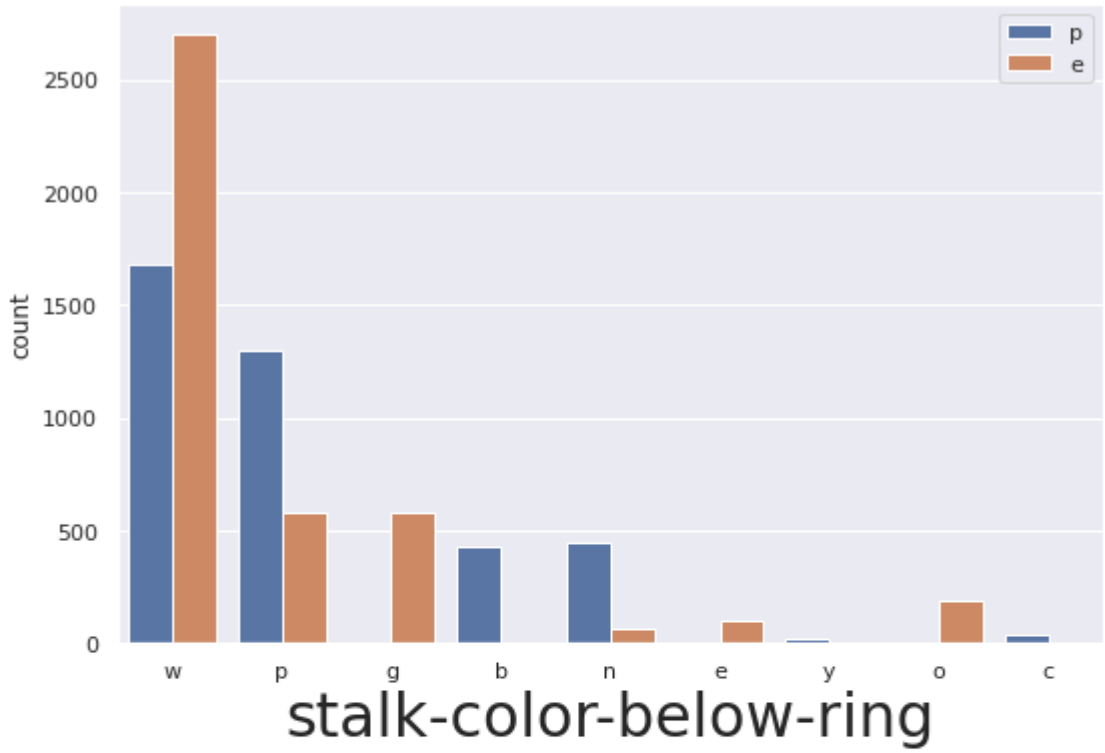
		stalk-surface-above-ring
stalk-surface-above-ring	class	
f	e	408
	p	144
k	e	144
	p	2228
s	e	3640
	p	1536
y	e	16
	p	8



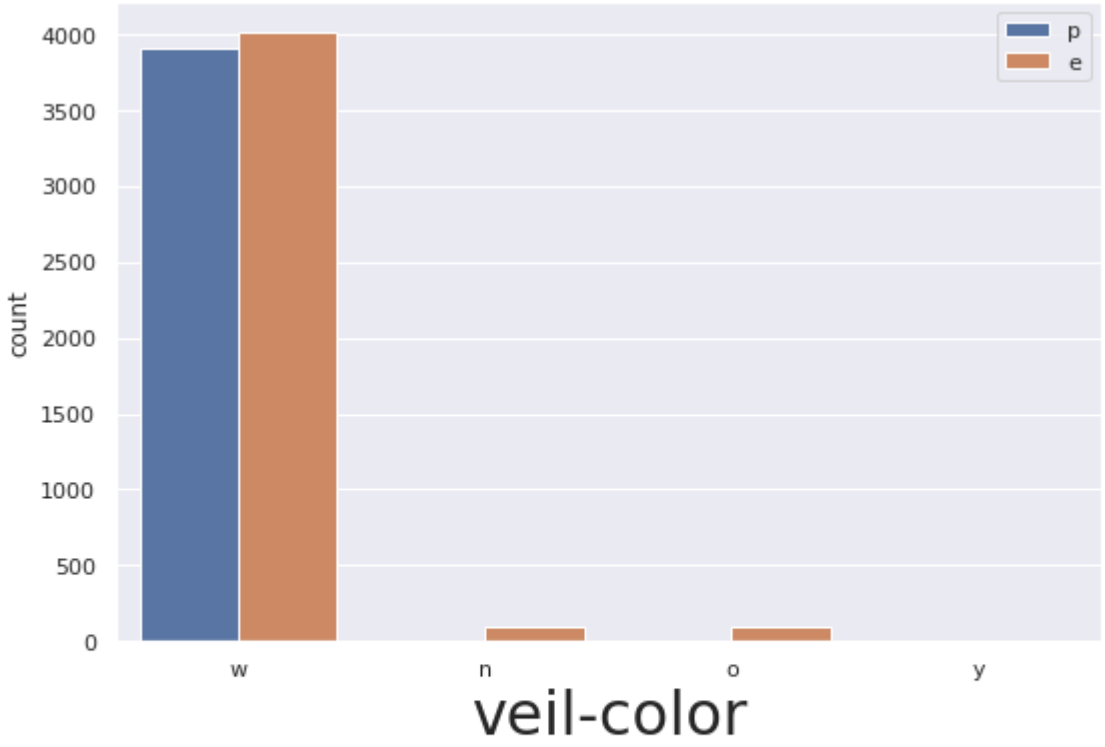
		stalk-surface-below-ring
stalk-surface-below-ring	class	
f	e	456
	p	144
k	e	144
	p	2160
s	e	3400
	p	1536
y	e	208
	p	76



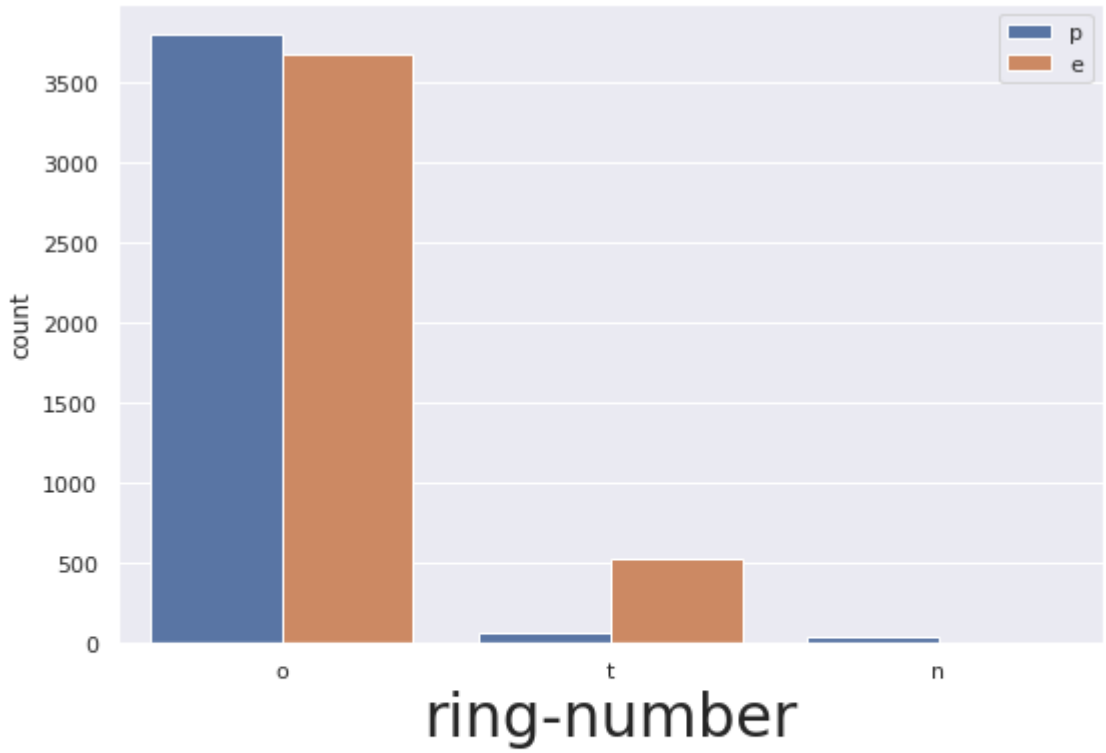
		stalk-color-above-ring
stalk-color-above-ring	class	
b	p	432
c	p	36
e	e	96
g	e	576
n	e	16
	p	432
o	e	192
p	e	576
	p	1296
w	e	2752
	p	1712
y	p	8



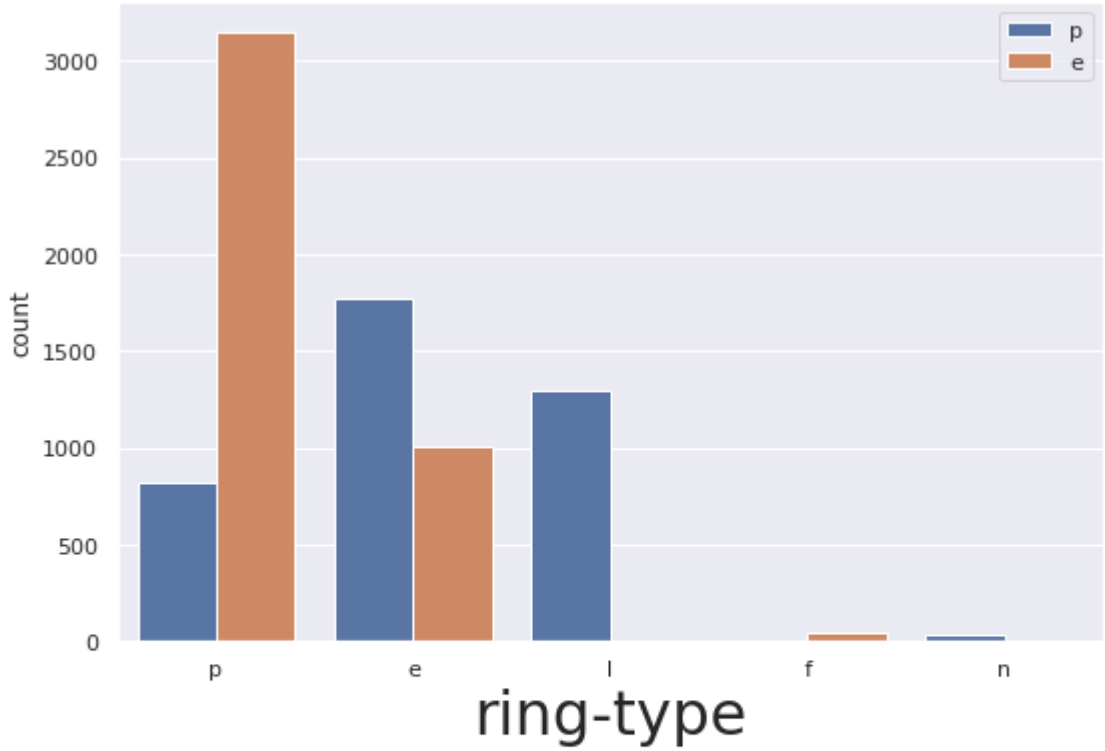
		stalk-color-below-ring
stalk-color-below-ring	class	
b	p	432
c	p	36
e	e	96
g	e	576
n	e	64
	p	448
o	e	192
p	e	576
	p	1296
w	e	2704
	p	1680
y	p	24



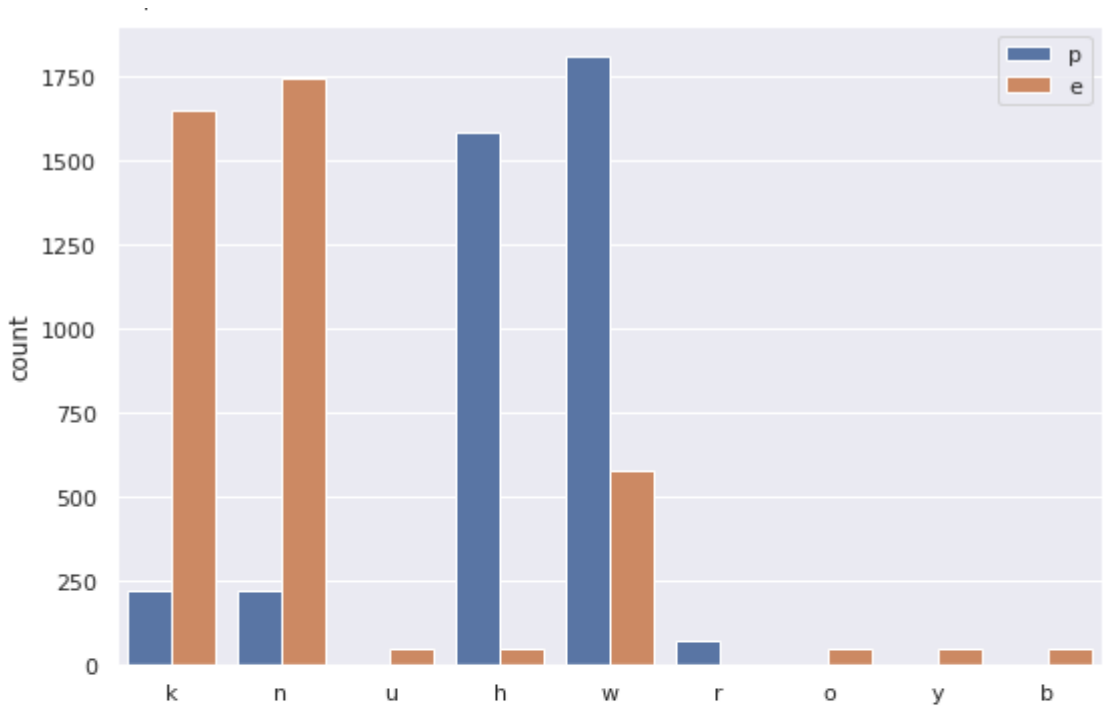
		veil-color
veil-color	class	
n	e	96
o	e	96
w	e	4016
	p	3908
y	p	8



		ring-number
ring-number	class	
n	p	36
o	e	3680
	p	3808
t	e	528
	p	72

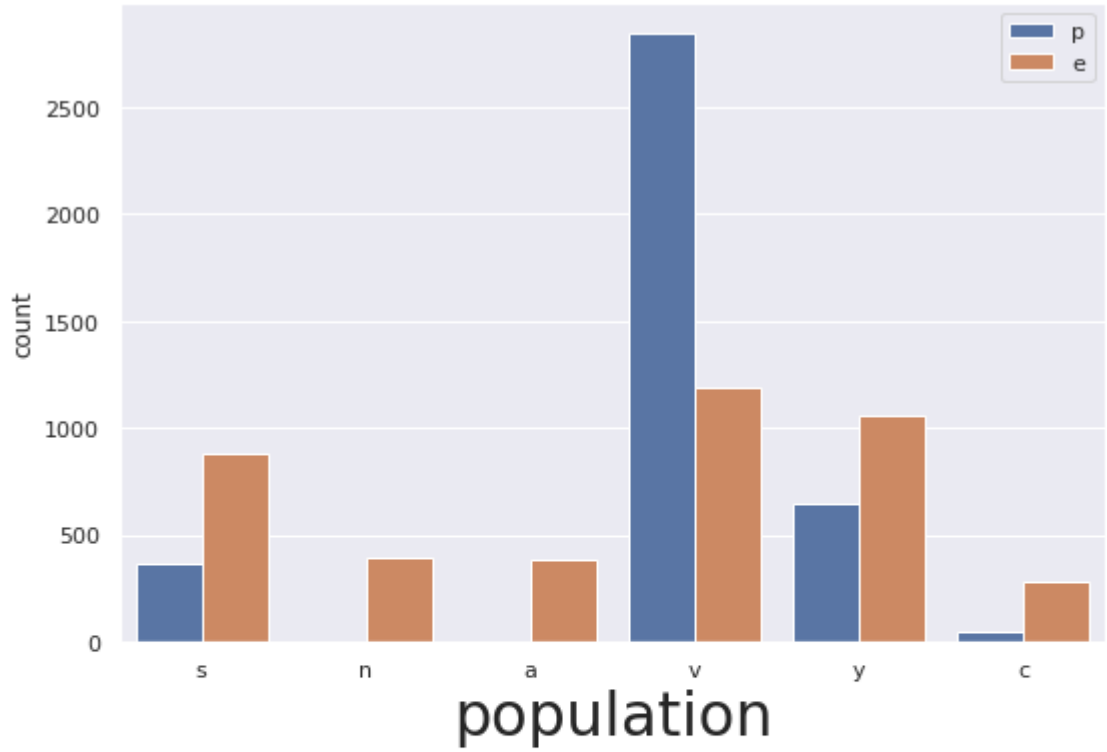


		ring-type
ring-type	class	
e	e	1008
	p	1768
f	e	48
l	p	1296
n	p	36
p	e	3152
	p	816

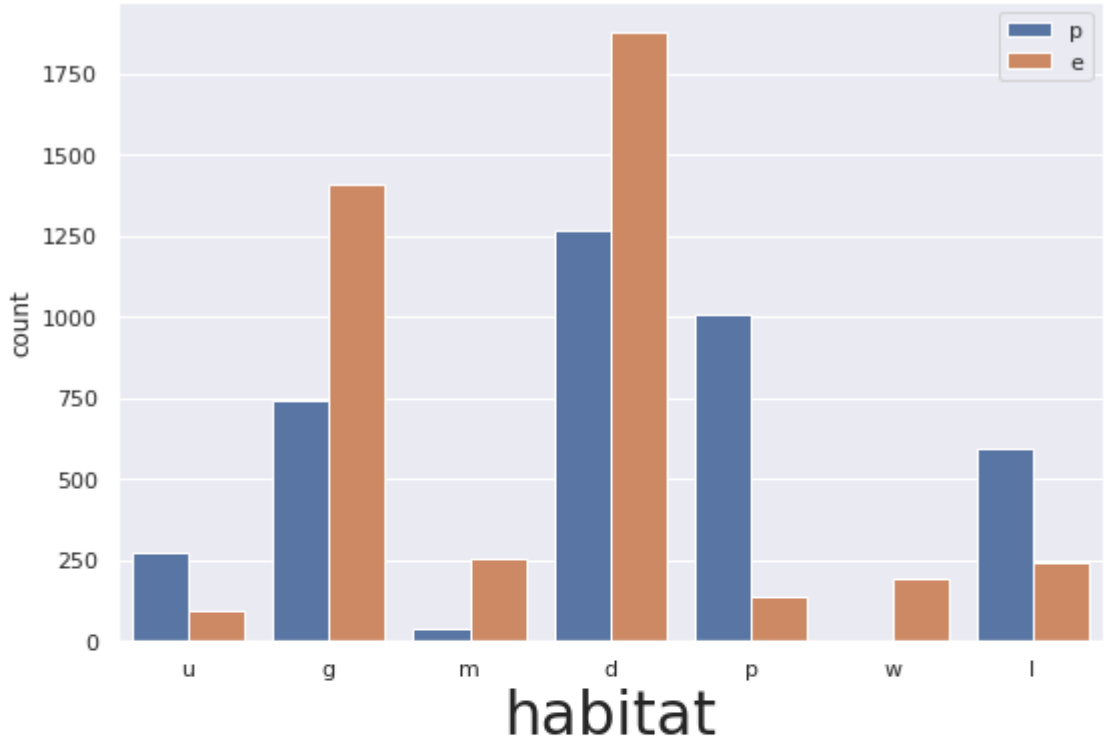


spore-print-color

		spore-print-color
spore-print-color	class	
b	e	48
h	e	48
	p	1584
k	e	1648
	p	224
n	e	1744
	p	224
o	e	48
r	p	72
u	e	48
w	e	576
	p	1812
y	e	48



		population
population	class	
a	e	384
c	e	288
	p	52
n	e	400
s	e	880
	p	368
v	e	1192
	p	2848
y	e	1064
	p	648



habitat	class	habitat
d	e	1880
	p	1268
g	e	1408
	p	740
l	e	240
	p	592
m	e	256
	p	36
p	e	136
	p	1008
u	e	96
	p	272
w	e	192

c. ACM