L'objectif de ce projet est d'apprendre, via le *maching learning*, à reconnaître si un champignon est comestible ou vénéneux en fonction de sa couleur, sa forme et la texture de sa cuticule.

Pour ce faire, on s'appuiera sur le site `https://ultimate-mushroom.com/`.

On ne se servira pas de l'IA développée dans ce projet comme oracle pour déterminer si on peut faire une omelette avec le résultat de sa cueillette !

Seule une connaissance humaine approfondie (par exemple, celle d'un pharmacien) est pour l'instant capable de trancher entre excellent comestible et champignon mortel.

Instructions

Ce projet est à réaliser impérativement en binôme : les projets individuels seront pénalisés. Seules les bibliothèques spécifiquement mentionnées dans le sujet sont autorisées.

Remise du code : Le projet devra être uploadé sur Eprel au plus tard le **12 avril 2024**.

Soutenance : Les soutenances auront lieu les 15, 16 et 17 avril. Chaque groupe disposera de 20 minutes de passage, divisées en 10 minutes de présentation et 10 minutes de questions. Nous interromprons la présentation au terme des 10 minutes : chronométrez-vous durant votre préparation pour être certain de ne pas dépasser.

Pour la soutenance : Sur fond violet, nous mettons en avant des sujets qu'il nous paraît intéressant d'aborder durant la soutenance. Ces suggestions sont facultatives, et tous les sujets ne pourront pas être abordés en 10 minutes – à vous de faire un choix. Rien ne vous empêche d'aborder d'autres points qui vous semblent digne d'intérêt.

1 Récupération des données en `python`

Tout projet de machine learning s'appuie sur un ensemble massif de données. L'objectif de cette première partie est la récupération de ces données, et leur stockage dans le format CSV.
Cette partie est à coder en `python`.

Question 1 Étudier les trois pages suivantes :

`https://ultimate-mushroom.com/poisonous/103-abortiporus-biennis.html`
`https://ultimate-mushroom.com/edible/1010-agaricus-albolutescens.html`
`https://ultimate-mushroom.com/inedible/452-byssonectria-terrestris.html`

En haut de chaque page (cf. l'encadré rouge des Figures 1, 2 et 3) est indiqué si le champignon est vénéneux (POISONOUS), comestible (EDIBLE) ou inoffensif mais sans intérêt gastronomique (INEDIBLE).

En utilisant les bibliothèques `requests` et `Beautiful soup` comme on l'a vu dans le TP2, écrire une fonction `comestible()` qui prend en entrée l'URL d'une page de champignon, et renvoie

- `"P"` si le champignon est POISONOUS,
- `"E"` si le champignon est EDIBLE,

- `"I"` si le champignon est INEDIBLE,
- `""` si la page est malformée.

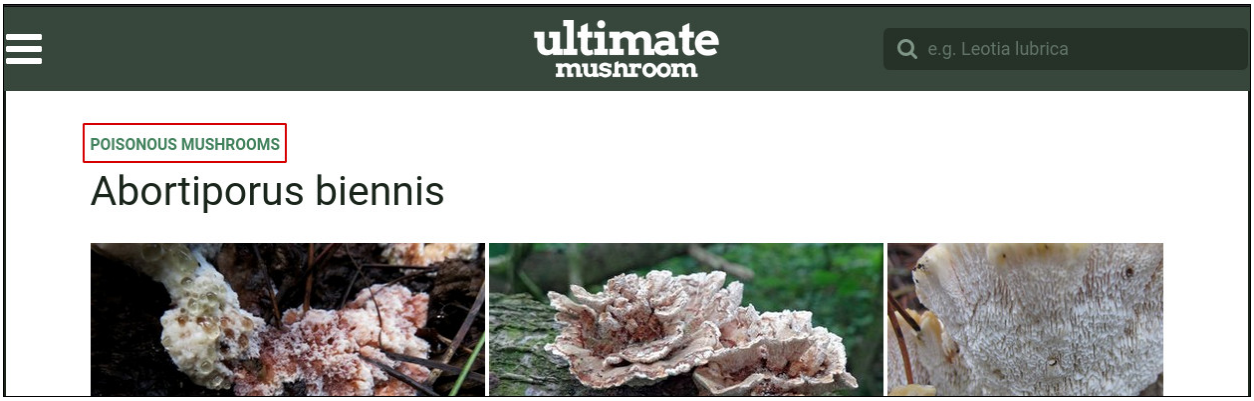

FIGURE 1 – L'Abortiporus biennis, à ne pas mettre dans son assiette !

FIGURE 2 – Un Agaricus albolutescens : direction la poêle.

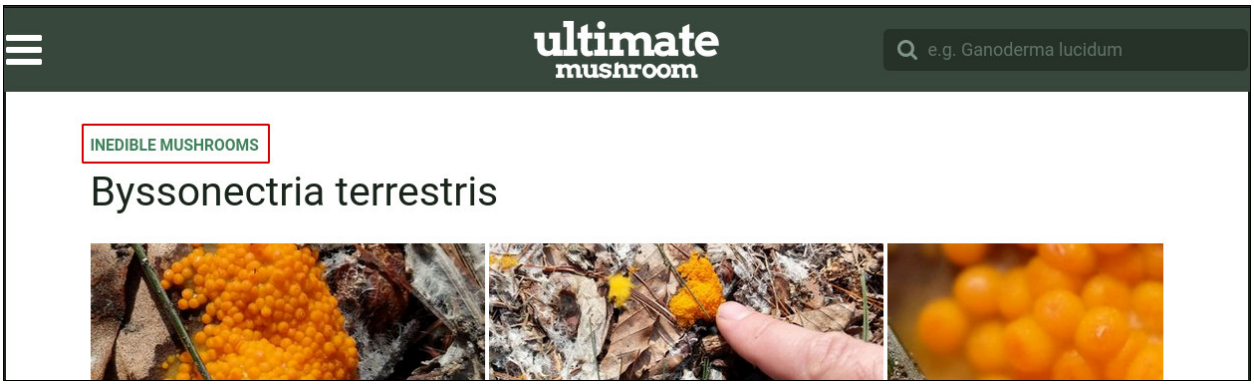

FIGURE 3 – Le Byssonectria terrestris, sans intérêt.

On va maintenant se pencher sur les caractéristiques physiques des champignons. Ce site

en répertorie trois : la couleur (COLOR), la forme (SHAPE) et la surface (SURFACE). Ces informations sont indiquées en bas de la page du champignon.

Question 2 Écrire une fonction `color()` qui attend l'URL d'une page de champignon, et renvoie une chaîne de caractères représentant la couleur du champignon (ou la chaîne vide si la page est malformée).

Pour la `https://ultimate-mushroom.com/poisonous/103-abortiporus-biennis.html`, on renverra donc la chaîne `"Pale"` (cf. Figure 4).

FIGURE 4 – Les caractéristiques physiques de l'Abortiporus biennis.

Sur la page `https://ultimate-mushroom.com/poisonous/1010-agaricus-albolutescens.html`, on remarquera que plusieurs couleurs sont parfois renseignées en même temps (cf. Figure 5). Dans ce cas, on mettra des tirets entre les différentes couleurs, et en renverra donc la chaîne `"White-Yellow"` dans le cas de cette page.

FIGURE 5 – Les caractéristiques physiques de l'Agaricus albolutescens.

Question 3 De la même manière, écrire deux fonctions `shape()` et `surface()` qui renvoient une chaîne de caractères représentant respectivement la forme et la surface du champignon.

On notera dans certains cas qu'une espace ou un tiret est présent dans les valeurs (par exemple "Cup Fungi", cf Figure 6). Dans ce cas, on enlèvera tout simplement l'espace ou le tiret : on renverra sur cet exemple la chaîne `"CupFungi"`.

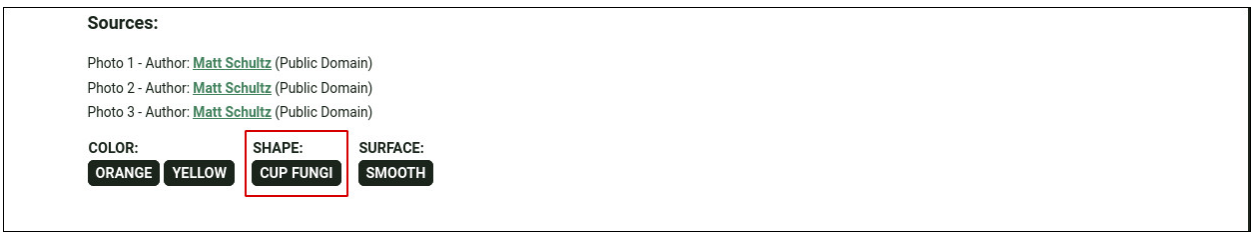

FIGURE 6 – Les caractéristiques physiques du Byssonectria terrestris.

Question 4 Écrire une fonction `csv()` qui attend l'URL de la page d'un champignon et renvoie une chaîne de caractères au format CSV (comma-separated values) représentant les caractéristiques du champignon, dans l'ordre TYPE, COLOR, SHAPE, SURFACE.

Par exemple, l'appel
`csv("https://ultimate-mushroom.com/inedible/452-byssonectria-terrestris.html")`
renverra la chaîne
`"I,Orange-Yellow,CupFungi,Smooth"`

Question 5 Coder un script qui va chercher l'ensemble des pages listées à l'adresse
`https://ultimate-mushroom.com/mushroom-alphabet.html`
et qui écrit dans un fichier `champignons.csv` l'ensemble des caractéristiques des champignons répertoriés par le site dans le format CSV spécifié à la Question 4, avec une ligne par champignon.

La première ligne de ce fichier sera donc

```
P,Pale,Polypore,Smooth
```

(sans guillemets) qui représente le premier champignon de la liste :
`https://ultimate-mushroom.com/poisonous/103-abortiporus-biennis.html`

2 Manipulation des données

Avant de pouvoir utiliser notre jeu de données dans le cadre d'un apprentissage automatique, il va nous falloir les analyser et les manipuler. Pour cela, on utilisera la bibliothèque `pandas`.

2.1 Préliminaires

Question 6 Installer `pandas`, et suivre le tutoriel en ligne pour se familiariser avec les notions de `DataFrame`, de `Series` et saisir les bases de leur manipulation.

Question 7 Ajouter une première ligne au fichier CSV pour nommer les colonnes. La première colonne devra être nommée `"Edible"`, la deuxième `"Color"`, la troisième `"Shape"` et la dernière `"Surface"`.

Après avoir importé le contenu du CSV dans un `DataFrame` nommé `champignons`, vérifier que votre jeu de données possède bien `1113` lignes et `4` attributs, ayant les bons noms.

La bibliothèque d'apprentissage automatique `scikit-learn`, que nous utiliserons dans la partie suivante, ne peut manipuler que des données numériques. Les attributs ayant pour l'instant comme valeurs des chaînes de caractères, il va falloir procéder à un certain nombre de conversions.

2.2 Colonne `"Edible"`

Commençons par la colonne `"Edible"`, qui sera la cible de notre apprentissage.

Question 8 Commencer par inspecter les données de cette colonne via la méthode `value_counts()`. On s'assurera de visualiser également les lignes vides (c'est-à-dire ayant la valeur `NA`).

On va affecter respectivement les valeurs numériques `0`, `1` et `2` aux valeurs `"E"`, `"I"` et `"P"`. Puisqu'on considérera dans la suite cette colonne comme catégorielle non-ordonnée (c'est-à-dire qu'on n'y verra que des valeurs distinctes sans relation), ce choix arbitraire n'aura pas d'impact.

Question 9 En utilisant la méthode `replace()`, remplacer respectivement les valeurs `"E"`, `"I"` et `"P"` par `0`, `1` et `2` dans la colonne `"Edible"` de `champignons`.

Vérifier sur quelques lignes de `champignons` que le résultat est bien celui attendu.

Il nous faut maintenant traiter les données manquantes, puisque `scikit-learn` n'accepte pas les `NA`.

Question 10 À l'aide de la méthode `fillna()`, remplacer les valeurs manquantes par des `-1`. Comparer le résultat obtenu via `value_counts()` avec celui obtenu dans la Question 8 pour vous assurer que votre colonne contient nombre attendu de chaque valeur.

2.3 Colonnes `"Shape"` et `"Surface"`

On va maintenant passer aux deux colonnes `"Shape"` et `"Surface"`. Pour ces colonnes, on va utiliser la méthode des variables indicatrices. Cela consiste, par exemple pour la colonne `"Shape"`, à créer une nouvelle colonne pour chaque valeur `v` possible. Une ligne possèdera un `1` dans cette colonne si et seulement si `v` apparaît dans son champ `"Shape"`. Un exemple minimal est donné en Figure 7. On notera qu'une valeur `NA` se traduira par des `0` dans chaque colonne indicatrice.

	Shape	···
0	Convex-Bellshaped	···
1	Convex-Flat	···
2	-	···
3	Bellshaped	···

(a) Avant

	Convex	**Bellshaped**	**Flat**	···
0	1	1	0	···
1	1	0	1	···
2	0	0	0	···
3	0	1	0	···

(b) Après

FIGURE 7 – Illustration de l'introduction de variables indicatrices

La bibliothèque `pandas` propose une fonction `get_dummies()` qui crée automatiquement des variables indicatrices pour une colonne. Malheureusement, cela ne fonctionne que quand la colonne contient des valeurs uniques. Dans notre cas, une même ligne peut avoir plusieurs valeurs, séparées par des tirets : on va donc devoir créer ces colonnes indicatrices manuellement.

On va commencer par s'occuper de la colonne `"Shape"`, et établir la liste de toutes les valeurs présentes dans cette colonne. On peut récupérer la liste de ces valeurs grâce à l'expression suivante :

```
pd.unique(champignons["Shape"].str.split("-").explode().dropna())
```

où `pd` est l'alias sous lequel on a importé la bibliothèque `pandas`.

Question 11 En consultant la documentation, comprendre ce que fait cette expression.

Question 12 En utilisant la liste ci-dessus, ajouter au `DataFrame` une nouvelle colonne (remplie de `0` et de `1` comme expliqué plus haut) pour chaque valeur que peut prendre la forme d'un champignon.

Pour ce faire, on pourra utiliser la méthode `str.contains()`. Cette méthode renvoie un booléen, qu'on transformera en entier via un appel à `astype(int)`.

Question 13 Procéder de la même manière pour la colonne `"Surface"`. On factorisera le code au maximum.

Supprimer les deux colonnes `"Shape"` et `"Surface"`, maintenant inutiles. À ce stade, votre `DataFrame` devrait être de taille `(1113,35)`.

2.4 Colonne `"Color"`

Plutôt que d'introduire des variables indicatrices pour la colonne `"Color"`, on va essayer de profiter de notre connaissance des couleurs pour améliorer notre modèle. Il parait en effet raisonnable de prendre en considération la proximité des couleurs sur le spectre. Pour cela, on va utiliser le format RGB, dans lequel une couleur est représentée par trois entiers compris entre `0` et `255` : le premier (`"R"`) pour le taux de rouge, le second (`"G"`) pour le taux de vert, et le dernier (`"B"`) indiquant le taux de bleu.

Question 14 En utilisant les méthodes vues plus haut, établir la liste des couleurs individuelles présentes dans notre jeu de données. Elles sont au nombre de `15`.

Question 15 Créer un `DataFrame` comportant `4` colonnes (une colonne `"Color"` contenant le nom de la couleur, et trois colonnes `"R"`, `"G"`, `"B"`) et `15` lignes (une pour chaque couleur présente dans le jeu de données). Pour savoir comment remplir cette table, on utilisera un logiciel ou un site web capable d'indiquer la décomposition RGB d'une couleur donnée.

On a maintenant le code RGB de toutes les couleurs individuelles présentes dans notre table. Il faut maintenant combiner ces couleurs, dans la mesure où certains champignons ont plusieurs couleurs. On commence par constater qu'un champignon a au maximum deux couleurs : en effet,

```
chamignons["Color"].str.split("-").str.len().max()
```

est évalué à `2`. Si un champignon possède deux couleurs, alors on décide que son `"R"` devra être la moyenne des `"R"` de ses deux couleurs ; idem pour `"G"` et `"B"`.

Question 16 Construire un `DataFrame colors` ayant une seule colonne, qui contient le nom de chaque combinaison unique de couleurs apparaissant parmi les champignons.

Question 17 En utilisant la fonction `merge()` (qui permet de faire une jointure entre deux tables) et la méthode `mean()` (qui permet de faire la moyenne de plusieurs valeurs – deux dans notre cas), compléter `colors` avec trois colonnes `"R"`, `"G"` et `"B"`. On pourra combiner les méthodes `str.split()` et `str.get()`.

Vérifier qu'on a bien obtenu le résultat attendu.

Question 18 Pour terminer, utiliser à nouveau `merge()` pour compléter `champignons` avec trois colonnes `"R"`, `"G"` et `"B"`. On supprimera la colonne `"Color"`, qui est maintenant inutile.

Il faut également donner des valeurs à la décomposition RGB des champignons dont le champ `"Color"` contenait `NA`. Utiliser `fillna()` pour que ces champignons aient `-255` dans chacune de leurs trois colonnes de couleur.

On a maintenant terminé de traiter les données. Notre `DataFrame` devrait maintenant être de format `(1113,37)` et ne plus contenir aucun `NA`.

3 Apprentissage et mise en forme

Nous sommes prêts à passer à l'apprentissage à proprement parler. Pour ce faire, nous allons utiliser la bibliothèque `scikit-learn`.

3.1 Premier modèle : Support Vector Machine (SVM)

Dans un premier temps, nous allons entraîner une *Support Vector Machine* sur nos données.

Question 19 Séparer les données en un jeu d'entraînement et un jeu de test, sur un base 75%/25%.

Question 20 Entraîner un modèle SVC sur le premier jeu.
Évaluer l'`accuracy_score` de ce modèle sur le jeu de test, puis détailler les résultats via une `confusion_matrix`.

Pour la soutenance : Comparer le score obtenu à celui qu'on aurait obtenu à l'aide du modèle qui prédit toujours le résultat majoritaire. En considérant la nature des données, est-il raisonnable d'espérer entraîner des modèles extrêmement fiables ?

Question 21 Reprendre la Question 20, en standardisant les colonnes dans un premier temps, via un `StandardScaler`.

Pour la soutenance : Comment expliquer la différence entre la précision obtenue avant `StandardScaler` et celle obtenue après standardisation ?

3.2 Deuxième modèle : Arbre de décision

Question 22 Reprendre les Questions 20 et 21 en entraînant cette fois un arbre de décision. On se limitera dans un premier temps à une profondeur 3.

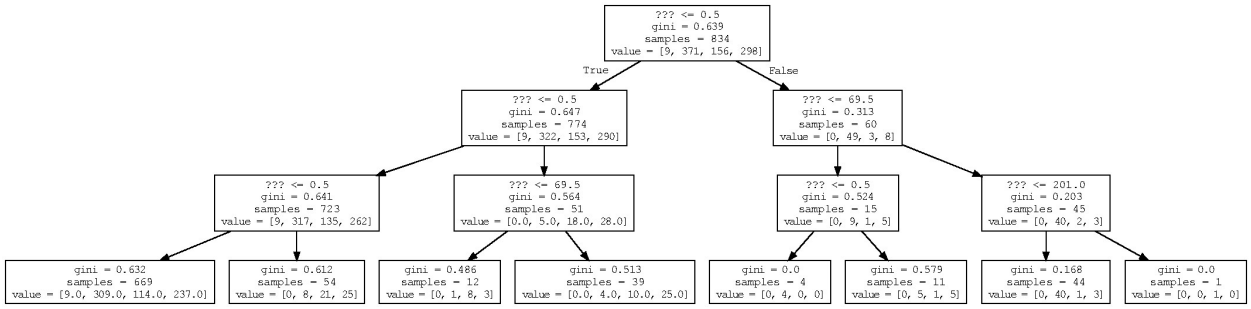

FIGURE 8 – Arbre de décision à profondeur 3. Ici, les noms des caractéristiques ont été masqués.

Question 23 Grâce à la fonction `export_graphviz`, donner une représentation graphique de l'arbre de décision entraîné dans la Question 22, comme illustré dans la Figure 8 (dans cette image, les noms de toutes les caractéristiques ont été remplacés par `"???"` – il faudra bien évidemment que leurs vrais noms figurent dans la représentation).

Pour la soutenance : Détailler ce que l'arbre de décision nous apprend sur les données.

3.3 Sauvegarde des modèles entraînés

Une fois un modèle entraîné, nous aimerions pouvoir le sauvegarder sur disque afin de nous en servir pour des prédictions futures, sans avoir besoin de le réentraîner en permanence.

Question 24 En suivant les instructions de `scikit-learn` concernant la persistance des modèles, sauvegarder chacun des deux modèles dans un fichier.

3.4 Accès aux données via le web

Pour terminer, nous allons mettre en place un service web minimal permettant de déterminer si un champignon est (supposément) vénéneux. Nous utiliserons pour cela `Node.js`.

Question 25 Télécharger `Node.js` ainsi que le framework `express`.

Question 26 Écrire un fichier HTML `formulaire.html` contenant un formulaire avec les champs suivants :

- trois champs textuels ; un pour chaque couleur RGB,
- une checkbox pour chaque `Shape` possible,
- une checkbox pour chaque `Surface` possible,
- un menu déroulant avec deux options intitulées `"SVM"` et `"Arbre de décision"`,
- un bouton de soumission.

Question 27 En suivant les exemples de routing `express`, déclarer une route pour la méthode `GET` à l'URL `"/form"`. En utilisant le module `fs` et sa méthode asynchrone `readFile()`, s'assurer que cette route renvoie le contenu de `formulaire.html`.
On n'oubliera pas de préciser le port d'écoute (par exemple `8080`) avec la ligne suivante

```
app.listen(8080, () => console.log("Listening..."));
```

Question 28 Déclarer une autre route pour la méthode `POST` à l'URL `"/rep"`, vers laquelle on est dirigé suite à la soumission du formulaire. Grâce à `child_process`, faites en sorte que cette route lance un script `python` qui charge le modèle (sauvegardé en Question 24) spécifié dans le formulaire, et le lance sur les caractéristiques précisées dans le formulaire. Cette route renverra le résultat prédit.