Prédiction du niveau calorique des recettes avec apprentissage automatique

XGBoost V1

Ce notebook utilise un classifieur pour prédire le niveau calorique des recettes (BAS/MOYEN/HAUT) basé sur les ingrédients et instructions, avec préprocessing NLP, XGBoost et interprétation SHAP.

Objectifs:

- Classifier les recettes en 3 niveaux caloriques (bas < 250, moyen 250-500, haut > 500)
- Utiliser Random Forest vs. XGBoost avec bonnes pratiques
- Préprocessing NLP des ingrédients et instructions
- Interprétation avec SHAP (explicabilité très importante dans la nutrition)

1. Imports

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import xgboost as xgb
from sklearn.model_selection import train_test_split, cross_val_score, RandomizedSearchCV, StratifiedKFold
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from scipy.sparse import hstack, csr_matrix
from collections import Counter
import shap
import re
import ast
```

```python
import warnings
warnings.filterwarnings('ignore')

# Thème sombre
plt.style.use('dark_background')
plt.rcParams['axes.unicode_minus'] = False

# Palette de couleurs
colors = ['#FF6B9D', '#4ECDC4', '#45B7D1', '#96CEB4', '#FECA57', '#FF9FF3', '#54A0FF', '#5F27CD', '#A8E6CF', '#FFD93D']

pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)

print("Libraries importées avec succès!")
```

```
Libraries importées avec succès!
```

```
/home/zeus/miniconda3/envs/cloudspace/lib/python3.12/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please up
date jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

2. Chargement et exploration rapide des données

In [2]:
```python
# Chargement des données
df = pd.read_csv('data/RAW_recipes.csv')
print(f"Forme du dataset: {df.shape}")
print(f"\nColonnes: {df.columns.tolist()}")
print(f"\nPremières lignes:")
df.head()
# Informations sur le dataset
df.info()
print("\nValeurs manquantes:")
print(df.isnull().sum())
```

```
Forme du dataset: (231637, 12)

Colonnes: ['name', 'id', 'minutes', 'contributor_id', 'submitted', 'tags', 'nutrition', 'n_steps', 'steps', 'description', 'ing
redients', 'n_ingredients']

Premières lignes:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 231637 entries, 0 to 231636
Data columns (total 12 columns):
 #   Column          Non-Null Count   Dtype 
---  ------          --------------   ----- 
 0   name            231636 non-null  object
 1   id              231637 non-null  int64 
 2   minutes         231637 non-null  int64 
 3   contributor_id  231637 non-null  int64 
 4   submitted       231637 non-null  object
 5   tags            231637 non-null  object
 6   nutrition       231637 non-null  object
 7   n_steps         231637 non-null  int64 
 8   steps           231637 non-null  object
 9   description     226658 non-null  object
 10  ingredients     231637 non-null  object
 11  n_ingredients   231637 non-null  int64 
dtypes: int64(5), object(7)
memory usage: 21.2+ MB

Valeurs manquantes:
name                 1
id                   0
minutes              0
contributor_id       0
submitted            0
tags                 0
nutrition            0
n_steps              0
steps                0
description       4979
ingredients          0
n_ingredients        0
dtype: int64
```

3. Préprocessing des données nutritionnelles

In [3]:
```
def parse_nutrition(nutrition_str):
    """Parse la colonne nutrition pour extraire les valeurs nutritionnelles"""
    try:
        # Convertir la chaîne en liste
        nutrition_list = ast.literal_eval(nutrition_str)
        return nutrition_list
    except:
        return [0, 0, 0, 0, 0, 0, 0]

# Appliquer le parsing
df['nutrition_parsed'] = df['nutrition'].apply(parse_nutrition)

# Extraire les valeurs nutritionnelles (l'ordre est: calories, total_fat, sugar, sodium, protein, saturated_fat, carbohydrates
nutrition_columns = ['calories', 'total_fat', 'sugar', 'sodium', 'protein', 'saturated_fat', 'carbohydrates']
for i, col in enumerate(nutrition_columns):
    df[col] = df['nutrition_parsed'].apply(lambda x: x[i] if len(x) > i else 0)

# Supprimer les valeurs aberrantes de calories (> 3000 ou < 0)
df = df[(df['calories'] >= 0) & (df['calories'] <= 3000)]

print(f"Statistiques des calories après nettoyage:")
print(df['calories'].describe())
```

```
Statistiques des calories après nettoyage:
count    228486.000000
mean        408.524812
std         384.645804
min           0.000000
25%         172.600000
50%         309.100000
75%         507.900000
max        2999.800000
Name: calories, dtype: float64
```

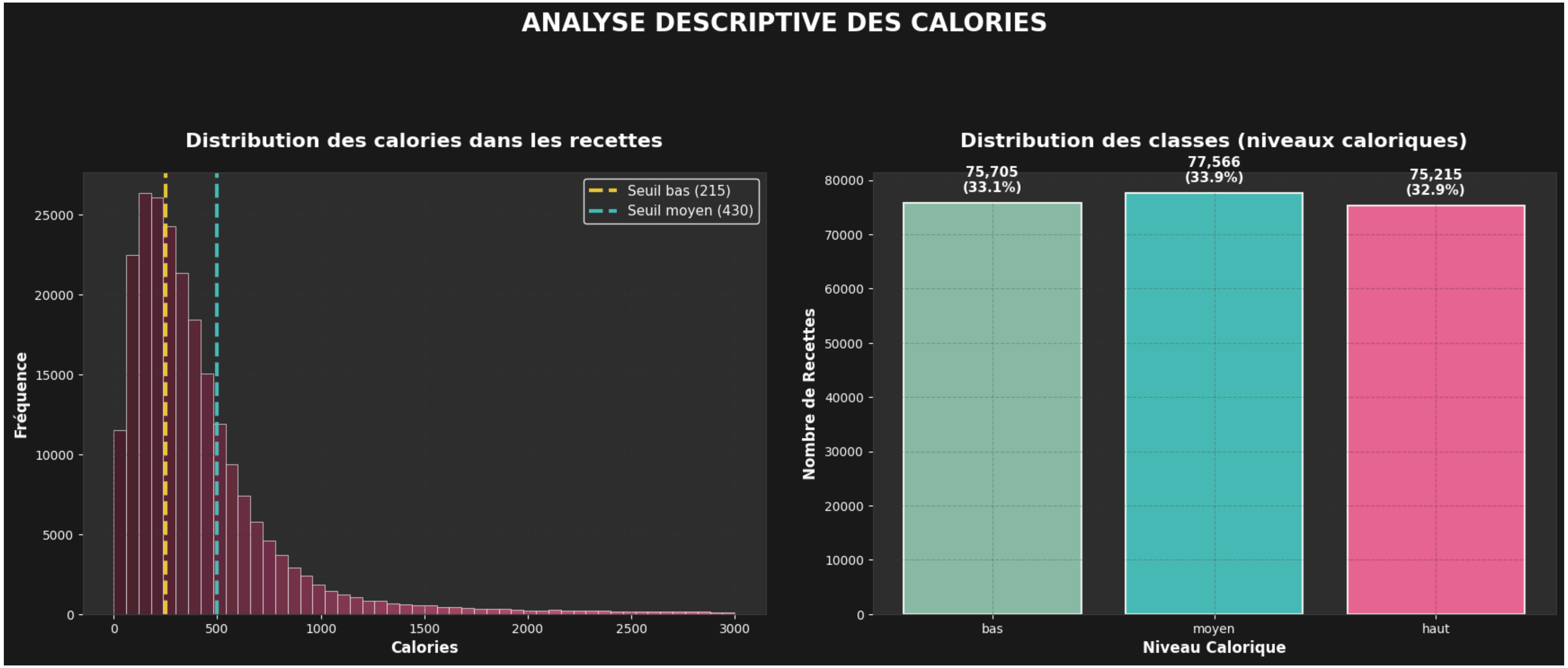
4. Analyse descriptive des calories

In [ ]:
```
# arrondir au supérieur
seuil_33 = int(df['calories'].quantile(0.33)) + 1
seuil_67 = int(df['calories'].quantile(0.67)) + 1

# seuils bas, moyen, haut (variable cible)
print(f"Seuil bas: 0-{seuil_33}, Seuil moyen: {seuil_33}-{seuil_67}, Seuil haut: {seuil_67}-{3000}")

def classify_calories_by_percentile(cal):
    if cal < seuil_33:
        return 'bas'
    elif cal <= seuil_67:
        return 'moyen'
    else:
        return 'haut'

# Recalculer avec des classes équilibrées (33.33% chacune)
df['calorie_level'] = df['calories'].apply(
    lambda x: classify_calories_by_percentile(x)
)

# Visualisation de la distribution
fig, axes = plt.subplots(1, 2, figsize=(18, 8))
fig.patch.set_facecolor('#1a1a1a')

# Distribution des calories
axes[0].set_facecolor('#2d2d2d')
n, bins, patches = axes[0].hist(df['calories'], bins=50, alpha=0.9,
                               edgecolor='white', linewidth=0.5)

for i, patch in enumerate(patches):
    base_color = np.array([255, 107, 157])  # #FF6B9D en RGB
    intensity = 0.3 + 0.7 * (i / len(patches))
    color = base_color * intensity / 255.0
    patch.set_facecolor(color)

axes[0].set_title('Distribution des calories dans les recettes', fontweight='bold',
                 fontsize=16, color='white', pad=20)
axes[0].set_xlabel('Calories', fontweight='bold', color='white', fontsize=12)
axes[0].set_ylabel('Fréquence', fontweight='bold', color='white', fontsize=12)
```

```
# Lignes de seuil
axes[0].axvline(x=250, color='#FFD93D', linestyle='--', linewidth=3,
               alpha=0.9, label=f'Seuil bas ({seuil_33})')
axes[0].axvline(x=500, color='#4ECDC4', linestyle='--', linewidth=3,
               alpha=0.9, label=f'Seuil moyen ({seuil_67})')

axes[0].tick_params(colors='white')
axes[0].grid(True, alpha=0.3, color='#404040', linestyle='--')

# Légende
legend = axes[0].legend(framealpha=0.9, facecolor='#2d2d2d',
                       edgecolor='white', fontsize=11)
for text in legend.get_texts():
    text.set_color('white')

for spine in axes[0].spines.values():
    spine.set_color('#404040')

# Distribution des niveaux caloriques
axes[1].set_facecolor('#2d2d2d')

calorie_counts = df['calorie_level'].value_counts()

ordered_levels = ['bas', 'moyen', 'haut']
ordered_counts = [calorie_counts.get(level, 0) for level in ordered_levels]

level_colors = ['#96CEB4', '#4ECDC4', '#FF6B9D']  # Vert, Cyan, Rose

bars = axes[1].bar(ordered_levels, ordered_counts,
                   color=level_colors, alpha=0.9,
                   edgecolor='white', linewidth=1.5)

axes[1].set_title('Distribution des classes (niveaux caloriques)',
                 fontweight='bold', fontsize=16, color='white', pad=20)
axes[1].set_xlabel('Niveau Calorique', fontweight='bold', color='white', fontsize=12)
axes[1].set_ylabel('Nombre de Recettes', fontweight='bold', color='white', fontsize=12)

for i, (bar, count) in enumerate(zip(bars, ordered_counts)):
    percentage = (count / sum(ordered_counts)) * 100
    axes[1].text(bar.get_x() + bar.get_width()/2,
                bar.get_height() + max(ordered_counts)*0.02,
```

```
                f'{count:,}\n({percentage:.1f}%)',
                ha='center', va='bottom', fontweight='bold',
                color='white', fontsize=11)

axes[1].tick_params(colors='white')
axes[1].grid(True, alpha=0.3, color='#404040', linestyle='--')

for spine in axes[1].spines.values():
    spine.set_color('#404040')

fig.suptitle('ANALYSE DESCRIPTIVE DES CALORIES',
             fontsize=20, fontweight='bold', color='white', y=0.98)

plt.tight_layout(pad=3.0, rect=[0, 0.03, 1, 0.95])
plt.show()

print("Distribution des niveaux caloriques:")
print(df['calorie_level'].value_counts())
print("\nPourcentages:")
print(df['calorie_level'].value_counts(normalize=True) * 100)
```

```
Seuil bas: 0-215, Seuil moyen: 215-430, Seuil haut: 430-3000
```




```
Distribution des niveaux caloriques:
calorie_level
moyen    77566
bas      75705
haut     75215
Name: count, dtype: int64

Pourcentages:
calorie_level
moyen    33.947813
bas      33.133321
haut     32.918866
Name: proportion, dtype: float64
```

5. Préprocessing NLP des ingrédients et instructions

In [5]:
```
def clean_text(text):
    """Nettoyage"""
    if pd.isna(text):
        return ""

    text = str(text).lower()

    # Supprimer crochets et guillemets
    text = re.sub(r"[\[\]'\"]", "", text)
    # Remplacer virgules par #
    text = re.sub(r',\s*', '#', text)
    # Garder les ingrédients composés avec _ ex: olive oil = olive_oil
    text = re.sub(r"\s+", " ", text)
    text = re.sub(r'\s+', '_', text)
    # Remettre espace à la place de #
    text = re.sub(r'#\s*', ' ', text)
    # Supprimer caractères spéciaux (sauf _)
    text = re.sub(r"[^a-zA-Z0-9_\s]", "", text)

    return text.strip()

def sort_ingredients(ingredients_text):
    """
    Trie les ingrédients par ordre alphabétique avant nettoyage
    """
    try:
        # Convertir la chaîne en liste
        ingredients_list = ast.literal_eval(ingredients_text)
        # Trier par ordre alphabétique
        sorted_ingredients = sorted(ingredients_list)
        # Retourner comme chaîne
        return str(sorted_ingredients)
    except:
        # Si échec, retourner tel quel
        return ingredients_text

# Test de la fonction de nettoyage
test_ingredient = "['chopped fresh spinach', 'tomato', 'olive oil', 'butter']"
print(f"\nTest de la fonction optimisée:")
print(f"Avant: {test_ingredient}")
```

```python
print(f"Après: {clean_text(test_ingredient)}")

# Trier et nettoyer les ingrédients
print(f"\nTri et nettoyage des ingrédients en cours...")
df['ingredients_sorted'] = df['ingredients'].apply(sort_ingredients)
df['ingredients_cleaned'] = df['ingredients_sorted'].apply(clean_text)

# Supprimer les recettes avec du texte vide
df = df[df['ingredients_cleaned'].str.len() > 10]

print(f"Nombre de recettes après nettoyage avancé: {len(df)}")
print("\nExemple de texte nettoyé et optimisé:")
print(df['ingredients_cleaned'].iloc[0][:200] + "...")

# Statistiques d'amélioration
print(f"\nStatistiques d'amélioration:")
word_counts = df['ingredients_cleaned'].apply(lambda x: len(x.split()))
print(f"Nombre moyen de mots par recette: {word_counts.mean():.1f}")
print(f"Nombre médian de mots par recette: {word_counts.median():.1f}")
print(f"Recettes avec moins de 5 mots: {(word_counts < 5).sum():,}")
print(f"Recettes avec 5-15 mots: {((word_counts >= 5) & (word_counts <= 15)).sum():,}")
print(f"Recettes avec plus de 15 mots: {(word_counts > 15).sum():,}")
```

```
Test de la fonction optimisée:
Avant: ['chopped fresh spinach', 'tomato', 'olive oil', 'butter']
Après: chopped_fresh_spinach tomato olive_oil butter

Tri et nettoyage des ingrédients en cours...
Nombre de recettes après nettoyage avancé: 228430

Exemple de texte nettoyé et optimisé:
butter honey mexican_seasoning mixed_spice olive_oil salt winter_squash...

Statistiques d'amélioration:
Nombre moyen de mots par recette: 9.1
Nombre médian de mots par recette: 9.0
Recettes avec moins de 5 mots: 21,187
Recettes avec 5-15 mots: 194,500
Recettes avec plus de 15 mots: 12,743
```

6. Extraction de features

In [6]:
```
protein_ingredients = {
    'eggs', 'egg', 'egg_whites', 'egg_yolks', 'hardboiled_eggs', 'egg_substitute',
    'chicken', 'cooked_chicken', 'chicken_breasts', 'boneless_skinless_chicken_breasts',
    'ground_beef', 'lean_ground_beef', 'beef', 'beef_brisket', 'beef_stew_meat',
    'ground_turkey', 'turkey', 'ground_pork', 'pork', 'pork_chops', 'pork_tenderloin',
    'lamb', 'sausage', 'smoked_sausage', 'italian_sausage', 'chicken_thighs',
    'chicken_drumsticks', 'chicken_breast_halves',
    'ham', 'deli_ham', 'prosciutto', 'bacon', 'cooked_bacon',
    'tofu', 'firm_tofu', 'extra_firm_tofu', 'soybeans', 'tempeh',
    'shrimp', 'large_shrimp', 'medium_shrimp', 'raw_shrimp', 'cooked_shrimp',
    'salmon', 'smoked_salmon', 'tuna', 'tuna_in_water',
    'crabmeat', 'lump_crabmeat', 'fish_fillets',
    'chickpeas', 'garbanzo_beans', 'black_beans', 'white_beans',
    'pinto_beans', 'kidney_beans', 'cannellini_beans', 'great_northern_beans',
    'refried_beans', 'lentils', 'baked_beans'
}

vegetable_ingredients = {
    'onion', 'onions', 'yellow_onion', 'white_onion', 'red_onion', 'sweet_onion', 'vidalia_onion',
    'garlic', 'garlic_cloves', 'garlic_clove', 'minced_garlic_clove', 'fresh_garlic',
    'carrots', 'carrot', 'baby_carrots',
    'potatoes', 'sweet_potatoes', 'red_potatoes', 'russet_potatoes', 'baking_potatoes',
    'celery', 'celery_ribs', 'celery_rib', 'celery_salt', 'celery_seed',
    'green_beans', 'fresh_green_beans', 'snap_peas',
    'green_pepper', 'green_peppers', 'green_bell_pepper', 'red_bell_pepper', 'yellow_bell_pepper', 'bell_pepper',
    'zucchini', 'eggplant', 'cabbage', 'red_cabbage', 'green_cabbage', 'napa_cabbage',
    'cauliflower', 'cauliflower_florets', 'broccoli', 'broccoli_florets', 'fresh_broccoli',
    'mushrooms', 'sliced_mushrooms', 'fresh_mushrooms', 'button_mushrooms', 'portabella_mushrooms',
    'asparagus', 'fresh_asparagus', 'asparagus_spears',
    'tomatoes', 'cherry_tomatoes', 'grape_tomatoes', 'roma_tomatoes', 'plum_tomatoes',
    'spinach', 'fresh_spinach', 'baby_spinach', 'baby_spinach_leaves', 'spinach_leaves', 'frozen_spinach',
    'lettuce', 'romaine_lettuce', 'iceberg_lettuce', 'lettuce_leaves', 'lettuce_leaf',
    'cucumber', 'cucumbers', 'english_cucumber', 'kale', 'chard', 'arugula',
    'leeks', 'leek', 'scallions', 'scallion', 'spring_onions', 'green_onions', 'green_onion'
}
```

```
spice_ingredients = {
    'salt', 'sea_salt', 'kosher_salt', 'seasoning_salt', 'table_salt',
    'black_pepper', 'ground_black_pepper', 'white_pepper', 'cracked_black_pepper',
    'paprika', 'smoked_paprika', 'sweet_paprika',
    'cayenne', 'cayenne_pepper', 'red_pepper_flakes', 'ground_red_pepper',
    'chili_powder', 'chili_flakes', 'chili_pepper', 'chili',
    'turmeric', 'ground_turmeric', 'turmeric_powder',
    'cumin', 'ground_cumin', 'cumin_powder', 'cumin_seed', 'cumin_seeds',
    'mustard', 'dry_mustard', 'prepared_mustard', 'mustard_powder', 'mustard_seeds',
    'garlic_powder', 'onion_powder', 'garlic_salt', 'onion_salt',
    'nutmeg', 'ground_nutmeg', 'cloves', 'ground_cloves',
    'cinnamon', 'ground_cinnamon', 'cinnamon_stick', 'cinnamon_sticks',
    'ginger', 'ground_ginger', 'gingerroot', 'fresh_ginger', 'crystallized_ginger',
    'bay_leaf', 'bay_leaves', 'allspice', 'ground_allspice',
    'cardamom', 'cardamom_pods', 'fennel_seed', 'anise', 'star_anise',
    'dried_oregano', 'oregano', 'oregano_leaves',
    'thyme', 'dried_thyme', 'fresh_thyme', 'thyme_leaves',
    'rosemary', 'dried_rosemary', 'fresh_rosemary',
    'basil', 'dried_basil', 'fresh_basil', 'fresh_basil_leaf',
    'parsley', 'fresh_parsley', 'dried_parsley', 'parsley_flakes',
    'sage', 'fresh_sage', 'dried_sage',
    'dill', 'dill_weed', 'dried_dill', 'dried_dill_weed',
    'mint', 'mint_leaf', 'mint_leaves',
    'marjoram', 'dried_marjoram',
    'tarragon', 'fresh_tarragon', 'dried_tarragon',
    'coriander', 'ground_coriander', 'coriander_seed', 'coriander_powder', 'coriander_leaves',
    'herbes_de_provence', 'italian_seasoning', 'poultry_seasoning', 'creole_seasoning', 'old_bay_seasoning'
}

grain_ingredients = {
    'flour', 'allpurpose_flour', 'whole_wheat_flour', 'white_flour',
    'unbleached_flour', 'plain_flour', 'cake_flour', 'bread_flour',
    'selfrising_flour', 'self_raising_flour',
    'cornmeal', 'cornflour', 'rice_flour', 'oatmeal', 'rolled_oats',
    'quick_oats', 'old_fashioned_oats', 'instant_oats',
    'pasta', 'spaghetti', 'penne_pasta', 'elbow_macaroni', 'macaroni', 'linguine',
    'fettuccine', 'rigatoni_pasta', 'orzo_pasta', 'bow_tie_pasta', 'wide_egg_noodles',
    'bread', 'whole_wheat_bread', 'baguette', 'french_bread', 'white_bread',
    'bisquick', 'bisquick_baking_mix',
    'couscous', 'quinoa', 'barley', 'pearl_barley',
    'rice', 'long_grain_rice', 'basmati_rice', 'brown_rice', 'white_rice',
```

```
    'tortillas', 'flour_tortillas', 'corn_tortillas',
    'crackers', 'saltine_crackers', 'graham_crackers'
}

fat_ingredients = {
    # Huiles et matières grasses
    'butter', 'unsalted_butter', 'salted_butter', 'clarified_butter', 'ghee', 'margarine',
    'shortening', 'vegetable_shortening', 'lard', 'crisco',
    'oil', 'olive_oil', 'extra_virgin_olive_oil', 'light_olive_oil',
    'vegetable_oil', 'canola_oil', 'corn_oil', 'sunflower_oil',
    'peanut_oil', 'sesame_oil', 'dark_sesame_oil', 'toasted_sesame_oil', 'grapeseed_oil',
    'coconut_oil', 'avocado_oil', 'palm_oil', 'salad_oil', 'cooking_oil',

    # Produits laitiers riches
    'cream', 'heavy_cream', 'heavy_whipping_cream', 'whipping_cream', 'double_cream',
    'sour_cream', 'light_sour_cream', 'fat_free_sour_cream', 'crème_fraîche', 'creme_fraiche',
    'clotted_cream', 'cream_cheese', 'light_cream_cheese', 'fat_free_cream_cheese',

    # Fromages (y compris dérivés allégés)
    'cheese', 'cheddar_cheese', 'sharp_cheddar_cheese', 'extrasharp_cheddar_cheese', 'mild_cheddar_cheese',
    'mozzarella_cheese', 'partskim_mozzarella_cheese', 'monterey_jack_cheese', 'colbymonterey_jack_cheese',
    'parmesan_cheese', 'fresh_parmesan_cheese', 'gruyere_cheese', 'swiss_cheese',
    'feta_cheese', 'goat_cheese', 'brie_cheese', 'blue_cheese', 'gorgonzola', 'romano_cheese',
    'ricotta_cheese', 'partskim_ricotta_cheese', 'mascarpone_cheese', 'velveeta_cheese',
    'asiago_cheese', 'provolone_cheese', 'fontina_cheese',

    # Noix, graines, et dérivés
    'nuts', 'almonds', 'slivered_almonds', 'sliced_almonds', 'ground_almonds',
    'walnuts', 'pecans', 'pecan_halves', 'cashews', 'peanuts', 'salted_peanuts',
    'macadamia_nuts', 'hazelnuts', 'pistachios',
    'pine_nuts', 'sunflower_seeds', 'pumpkin_seeds', 'sesame_seeds', 'chia_seeds', 'flax_seed_meal',

    # Tartinables
    'peanut_butter', 'creamy_peanut_butter', 'crunchy_peanut_butter', 'almond_butter',
    'tahini', 'nutella', 'chocolate_spread',

    # Fruits gras
    'avocado', 'avocados', 'olives', 'black_olives', 'green_olives', 'kalamata_olives', 'olive',
    'coconut', 'shredded_coconut', 'flaked_coconut', 'sweetened_flaked_coconut',
    'desiccated_coconut', 'coconut_milk', 'light_coconut_milk', 'unsweetened_coconut_milk', 'coconut_cream',
```

```
    # Viandes grasses
    'bacon', 'cooked_bacon', 'sausage', 'smoked_sausage', 'chorizo_sausage',
    'salami', 'pepperoni', 'ham', 'deli_ham', 'prosciutto',
    'ribeye', 'pork_belly', 'duck', 'goose', 'foie_gras',

    # Poissons gras
    'salmon', 'smoked_salmon', 'mackerel', 'trout', 'sardines', 'anchovies', 'herring',

    # Œufs et sauces riches
    'eggs', 'egg', 'egg_yolks', 'egg_yolk', 'mayonnaise', 'light_mayonnaise',
    'aioli', 'miracle_whip',

    # Lait entier et dérivés
    'whole_milk', 'milk', 'full_fat_milk', 'evaporated_milk', 'condensed_milk', 'sweetened_condensed_milk',
    'full_fat_yogurt', 'greek_yogurt', 'yogurt', 'plain_yogurt', 'fruit_yogurt', 'flavored_yogurt', 'labneh'
}

sugar_ingredients = {
    # Sucres
    'sugar', 'white_sugar', 'brown_sugar', 'light_brown_sugar', 'dark_brown_sugar',
    'granulated_sugar', 'powdered_sugar', 'icing_sugar', 'confectioners_sugar',
    'superfine_sugar', 'caster_sugar', 'coconut_sugar',

    # Sirops
    'honey', 'liquid_honey', 'raw_honey', 'maple_syrup', 'pure_maple_syrup', 'golden_syrup',
    'agave', 'agave_nectar', 'molasses', 'blackstrap_molasses',
    'corn_syrup', 'light_corn_syrup', 'simple_syrup', 'glucose_syrup', 'barley_syrup',

    # Chocolat et cacao
    'chocolate', 'dark_chocolate', 'milk_chocolate', 'white_chocolate',
    'chocolate_chips', 'semisweet_chocolate_chips', 'white_chocolate_chips', 'bittersweet_chocolate',
    'unsweetened_chocolate', 'chocolate_syrup', 'cocoa', 'cocoa_powder', 'unsweetened_cocoa_powder',

    # Fruits secs sucrés
    'dates', 'pitted_dates', 'prunes', 'raisins', 'golden_raisin', 'currants',
    'dried_cranberries', 'dried_cherries', 'dried_apricots', 'dried_apricot', 'dried_fruit',

    # Fruits frais naturellement sucrés
    'apple', 'apples', 'banana', 'bananas', 'pear', 'pears', 'orange', 'oranges',
    'grapes', 'granny_smith_apples', 'pineapple', 'pineapple_chunks', 'mango', 'mangoes',
    'papaya', 'kiwi', 'peach', 'peaches', 'nectarine', 'cherries', 'strawberry',
```

```
    'strawberries', 'raspberries', 'fresh_raspberries', 'blueberries', 'fresh_blueberries',
    'blackberries', 'berries', 'fruit', 'watermelon', 'cantaloupe', 'mandarin_oranges',

    # Lait sucré
    'condensed_milk', 'sweetened_condensed_milk', 'evaporated_milk', 'chocolate_milk', 'flavored_milk',

    # Produits sucrés
    'cake', 'cupcake', 'brownie', 'cookies', 'biscuit', 'muffin', 'pastry',
    'croissant', 'donut', 'pudding', 'custard', 'fudge', 'toffee', 'caramel',
    'ice_cream', 'vanilla_ice_cream', 'sorbet', 'jello', 'gelatin', 'apple_butter', 'marshmallows',
    'mini_marshmallows', 'whipped_topping', 'frosting', 'icing', 'instant_vanilla_pudding',

    # Confitures, pâtes sucrées
    'jam', 'jelly', 'marmalade', 'preserves', 'fruit_spread',
    'lemon_curd', 'dulce_de_leche', 'apricot_preserves', 'apricot_jam', 'raspberry_jam', 'cherry_pie_filling',

    # Jus sucrés et smoothies
    'fruit_juice', 'apple_juice', 'orange_juice', 'pineapple_juice', 'grape_juice',
    'cranberry_juice', 'pomegranate_juice', 'smoothie', 'sweetened_tea', 'frozen_orange_juice_concentrate'
}

drink_ingredients = {
    # Eaux & laits
    'water', 'cold_water', 'warm_water', 'boiling_water', 'ice', 'ice_cubes', 'crushed_ice',
    'milk', 'whole_milk', 'skim_milk', 'nonfat_milk', 'lowfat_milk', '2_lowfat_milk', '1_lowfat_milk',
    'almond_milk', 'soy_milk', 'soymilk', 'rice_milk', 'coconut_milk', 'light_coconut_milk',
    'evaporated_milk', 'condensed_milk', 'sweetened_condensed_milk',

    # Jus
    'orange_juice', 'apple_juice', 'grape_juice', 'cranberry_juice',
    'pineapple_juice', 'pomegranate_juice', 'lemon_juice', 'lime_juice',
    'vegetable_juice', 'carrot_juice', 'tomato_juice', 'fresh_orange_juice', 'fruit_juice',

    # Boissons chaudes
    'coffee', 'brewed_coffee', 'instant_coffee', 'instant_coffee_granules',
    'tea', 'black_tea', 'green_tea', 'herbal_tea', 'matcha', 'chai',
    'hot_chocolate', 'cocoa', 'cocoa_powder', 'milk_foam',

    # Sirops / édulcorants pour boissons
    'honey', 'agave', 'maple_syrup', 'corn_syrup', 'simple_syrup', 'grenadine', 'barley_syrup',
    'sugar', 'brown_sugar', 'light_brown_sugar', 'sweetened_tea',
```

```
    # Smoothie / milkshake add-ons
    'banana', 'berries', 'blueberries', 'strawberries', 'avocado',
    'yogurt', 'greek_yogurt', 'ice_cream', 'fruit_yogurt', 'protein_powder', 'spinach', 'kale',

    # Alcools
    'vodka', 'rum', 'light_rum', 'dark_rum', 'whiskey', 'brandy', 'bourbon',
    'tequila', 'triple_sec', 'amaretto', 'grand_marnier', 'vermouth', 'cognac',
    'champagne', 'wine', 'red_wine', 'white_wine', 'dry_white_wine', 'dry_red_wine',
    'beer', 'sake', 'mirin',

    # Boissons industrielles
    'cola', 'soda', 'root_beer', 'ginger_ale', 'tonic', 'energy_drink', 'sports_drink',
    'lemonade', 'club_soda'
}

def extract_advanced_features(df):
    """Extraire des features avancées pour la prédiction de calories"""

    def count_ingredients_by_category(row, ingredient_list):
        """
        Fonction générique pour compter les ingrédients d'une catégorie donnée

        Args:
            row: Ligne du DataFrame avec 'ingredients_cleaned'
            ingredient_list: Set/liste des ingrédients à rechercher

        Returns:
            int: Nombre d'ingrédients de cette catégorie trouvés
        """
        ingredients_text = str(row['ingredients_cleaned']).lower().split()
        count = 0

        for ingredient in ingredients_text:
            # Recherche exacte d'abord (plus rapide)
            if ingredient in ingredient_list:
                count += 1
            else:
                # Recherche de sous-chaînes (pour "olive_oil" dans "extra_virgin_olive_oil")
                for target_ing in ingredient_list:
                    if target_ing in ingredient:
```

```
                        count += 1
                        break

        return count

    print("Extraction des features avancées...")

    # Compter les ingrédients par catégorie
    df['nb_fat'] = df.apply(lambda row: count_ingredients_by_category(row, fat_ingredients), axis=1)
    df['nb_sugar'] = df.apply(lambda row: count_ingredients_by_category(row, sugar_ingredients), axis=1)
    df['nb_drink'] = df.apply(lambda row: count_ingredients_by_category(row, drink_ingredients), axis=1)
    df['nb_protein'] = df.apply(lambda row: count_ingredients_by_category(row, protein_ingredients), axis=1)
    df['nb_vegetable'] = df.apply(lambda row: count_ingredients_by_category(row, vegetable_ingredients), axis=1)
    df['nb_grain'] = df.apply(lambda row: count_ingredients_by_category(row, grain_ingredients), axis=1)
    df['nb_spice'] = df.apply(lambda row: count_ingredients_by_category(row, spice_ingredients), axis=1)

    epsilon = 1e-6

    # Ratios basiques
    df['fat_sugar_ratio'] = df['nb_fat'] / (df['nb_sugar'] + epsilon)
    df['fat_ratio'] = df['nb_fat'] / (df['n_ingredients'] + epsilon)
    df['sugar_ratio'] = df['nb_sugar'] / (df['n_ingredients'] + epsilon)
    df['drink_ratio'] = df['nb_drink'] / (df['n_ingredients'] + epsilon)
    df['protein_ratio'] = df['nb_protein'] / (df['n_ingredients'] + epsilon)
    df['vegetable_ratio'] = df['nb_vegetable'] / (df['n_ingredients'] + epsilon)
    df['grain_ratio'] = df['nb_grain'] / (df['n_ingredients'] + epsilon)
    df['spice_ratio'] = df['nb_spice'] / (df['n_ingredients'] + epsilon)

    print("Features avancées extraites avec succès!")

    return df

# Appliquer l'extraction de features
df = extract_advanced_features(df)

print("Features avancées extraites:")

print("- nb_fat: nombre d'ingrédients gras")
print("- nb_sugar: nombre d'ingrédients sucrés")
print("- nb_drink: nombre d'ingrédients de boisson")
print("- nb_protein: nombre d'ingrédients protéines")
```

```python
print("- nb_vegetable: nombre d'ingrédients légumes")
print("- nb_grain: nombre d'ingrédients céréales")
print("- nb_spice: nombre d'ingrédients épices")

print("- fat_ratio: ratio d'ingrédients gras")
print("- sugar_ratio: ratio d'ingrédients sucrés")
print("- drink_ratio: ratio d'ingrédients de boisson")
print("- protein_ratio: ratio d'ingrédients protéinés")
print("- vegetable_ratio: ratio d'ingrédients légumes")
print("- grain_ratio: ratio d'ingrédients céréales")
print("- spice_ratio: ratio d'ingrédients épices")

df.head()
```

```
Extraction des features avancées...
Features avancées extraites avec succès!
Features avancées extraites:
- nb_fat: nombre d'ingrédients gras
- nb_sugar: nombre d'ingrédients sucrés
- nb_drink: nombre d'ingrédients de boisson
- nb_protein: nombre d'ingrédients protéines
- nb_vegetable: nombre d'ingrédients légumes
- nb_grain: nombre d'ingrédients céréales
- nb_spice: nombre d'ingrédients épices
- fat_ratio: ratio d'ingrédients gras
- sugar_ratio: ratio d'ingrédients sucrés
- drink_ratio: ratio d'ingrédients de boisson
- protein_ratio: ratio d'ingrédients protéinés
- vegetable_ratio: ratio d'ingrédients légumes
- grain_ratio: ratio d'ingrédients céréales
- spice_ratio: ratio d'ingrédients épices
```

Out[6]:

	name	id	minutes	contributor_id	submitted	tags	nutrition	n_steps	steps	description	ingredients	n_ingred
0	arriba baked winter squash mexican style	137739	55	47892	2005-09-16	['60-minutes-or-less', 'time-to-make', 'course...	[51.5, 0.0, 13.0, 0.0, 2.0, 0.0, 4.0]	11	['make a choice and proceed with recipe', 'dep...	autumn is my favorite time of year to cook! th...	['winter squash', 'mexican seasoning', 'mixed ...	
1	a bit different breakfast pizza	31490	30	26278	2002-06-17	['30-minutes-or-less', 'time-to-make', 'course...	[173.4, 18.0, 0.0, 17.0, 22.0, 35.0, 1.0]	9	['preheat oven to 425 degrees f', 'press dough...	this recipe calls for the crust to be prebaked...	['prepared pizza crust', 'sausage patty', 'egg...	
2	all in the kitchen chili	112140	130	196586	2005-02-25	['time-to-make', 'course', 'preparation', 'mai...	[269.8, 22.0, 32.0, 48.0, 39.0, 27.0, 5.0]	6	['brown ground beef in large pot', 'add choppe...	this modified version of 'mom's' chili was a h...	['ground beef', 'yellow onions', 'diced tomato...	
3	alouette potatoes	59389	45	68585	2003-04-14	['60-minutes-or-less', 'time-to-make', 'course...	[368.1, 17.0, 10.0, 2.0, 14.0, 8.0, 20.0]	11	['place potatoes in a large pot of lightly sal...	this is a super easy, great tasting, make ahea...	['spreadable cheese with garlic and herbs', 'n...	
4	amish tomato ketchup for canning	44061	190	41706	2002-10-25	['weeknight', 'time-to-make', 'course', 'main-...	[352.9, 1.0, 337.0, 23.0, 3.0, 0.0, 28.0]	5	['mix all ingredients& boil for 2 1 / 2 hours ...	my dh's amish mother raised him on this recipe...	['tomato juice', 'apple cider vinegar', 'sugar...	

7. Visualisations pour comprendre la répartition des features

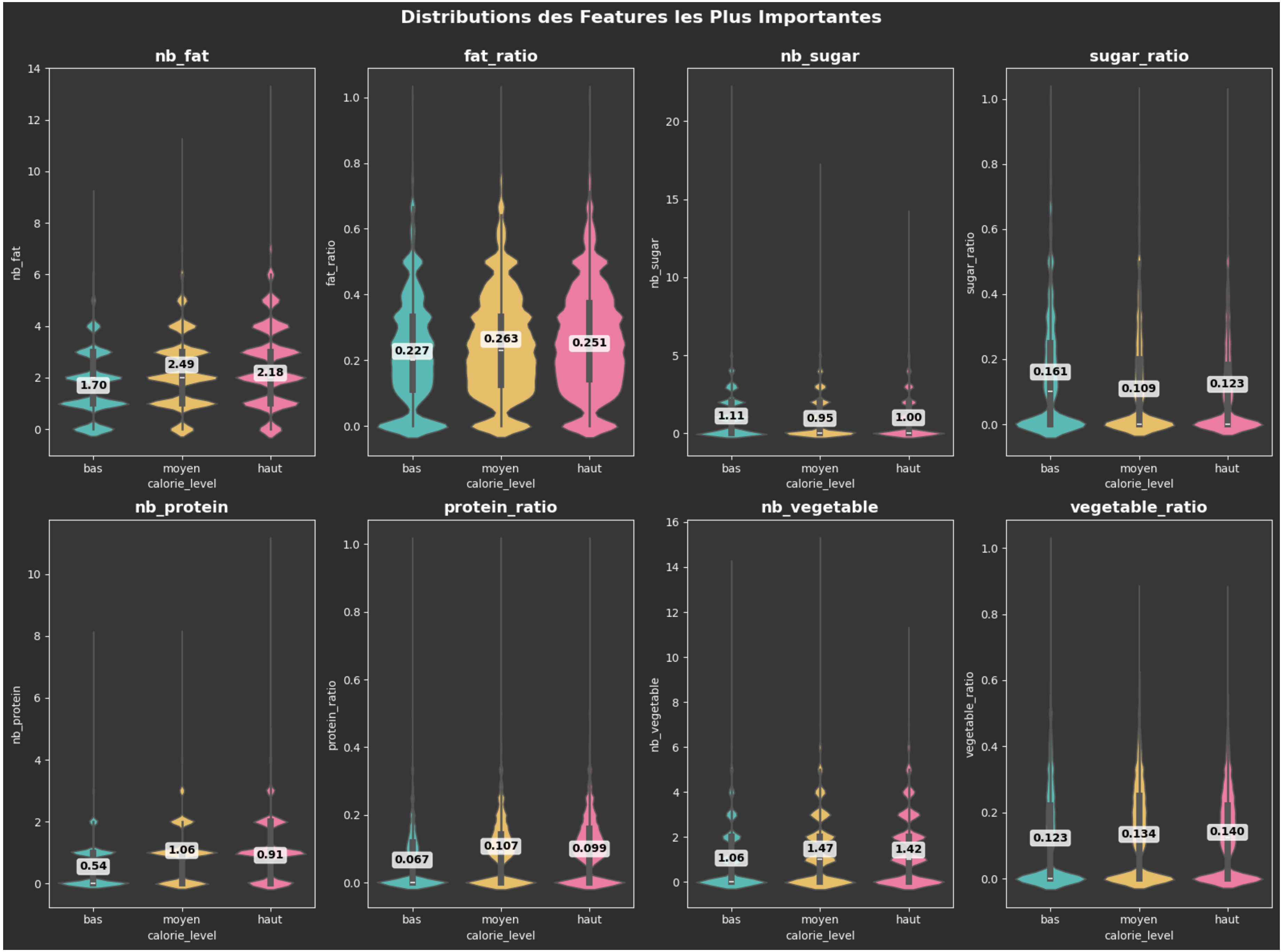
In [ ]:
```
plt.style.use('dark_background')
import matplotlib.colors as mcolors

class_colors = {'bas': '#4ECDC4', 'moyen': '#FECA57', 'haut': '#FF6B9D'}

# Paires de features (compteur + ratio correspondant)
feature_pairs = [
    ('nb_fat', 'fat_ratio'),
    ('nb_sugar', 'sugar_ratio'),
    ('nb_protein', 'protein_ratio'),
    ('nb_vegetable', 'vegetable_ratio'),
    ('nb_grain', 'grain_ratio'),
    ('nb_drink', 'drink_ratio'),
    ('nb_spice', 'spice_ratio')
]

# ================================
# FIGURE 1: VIOLIN PLOTS
# ================================
fig1 = plt.figure(figsize=(16, 12))
fig1.patch.set_facecolor('#2F2F2F')

# Calculer d'abord les scores pour prendre les plus importantes
features_all = [pair[0] for pair in feature_pairs] + [pair[1] for pair in feature_pairs]
means_by_class = df.groupby('calorie_level')[features_all].mean()
discrimination_score = means_by_class.std(axis=0).sort_values(ascending=False)

# Identifier les 4 paires les plus importantes
important_features = discrimination_score.head(8).index.tolist()
important_pairs = []
for count_feat, ratio_feat in feature_pairs:
    if count_feat in important_features or ratio_feat in important_features:
        important_pairs.append((count_feat, ratio_feat))
important_pairs = important_pairs[:4]  # Garder seulement les 4 premières

for i, (count_feature, ratio_feature) in enumerate(important_pairs):
    row = i // 2 + 1  # 2 lignes
    col = (i % 2) * 2 + 1  # 2 colonnes de 2 subplots

    # Subplot pour le compteur
```

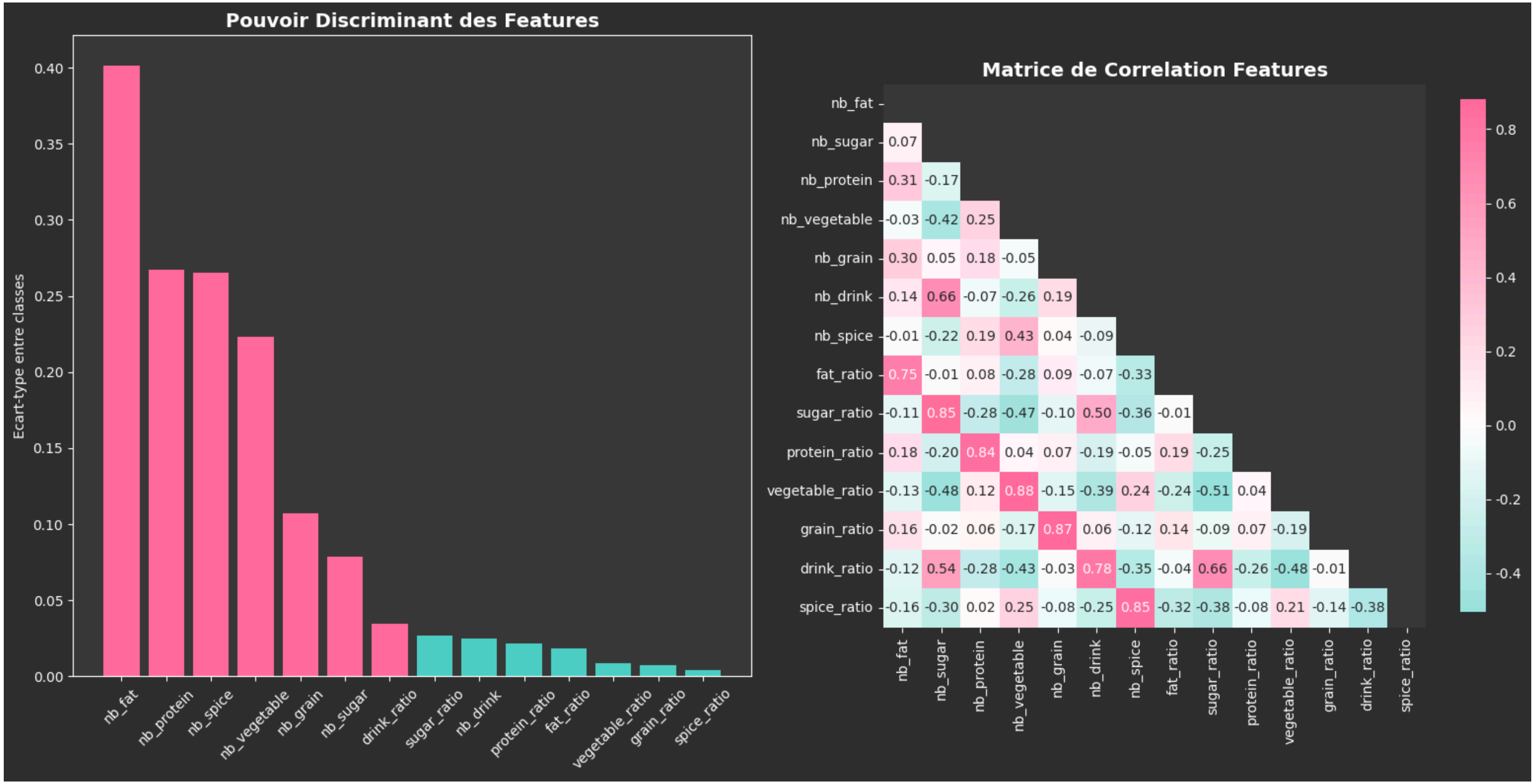
```
    ax1 = plt.subplot(2, 4, (row-1)*4 + col)
    ax1.set_facecolor('#3A3A3A')
    sns.violinplot(data=df, x='calorie_level', y=count_feature, palette=class_colors, ax=ax1)
    plt.title(f'{count_feature}', fontsize=14, fontweight='bold')

    # Ajouter les moyennes
    means = df.groupby('calorie_level')[count_feature].mean()
    for j, (level, mean_val) in enumerate(means.items()):
        plt.text(j, mean_val, f'{mean_val:.2f}', ha='center', va='center',
                fontweight='bold', color='black',
                bbox=dict(boxstyle="round,pad=0.2", facecolor='white', alpha=0.8))

    # Subplot pour le ratio correspondant
    ax2 = plt.subplot(2, 4, (row-1)*4 + col + 1)
    ax2.set_facecolor('#3A3A3A')
    sns.violinplot(data=df, x='calorie_level', y=ratio_feature, palette=class_colors, ax=ax2)
    plt.title(f'{ratio_feature}', fontsize=14, fontweight='bold')

    # Ajouter les moyennes
    means = df.groupby('calorie_level')[ratio_feature].mean()
    for j, (level, mean_val) in enumerate(means.items()):
        plt.text(j, mean_val, f'{mean_val:.3f}', ha='center', va='center',
                fontweight='bold', color='black',
                bbox=dict(boxstyle="round,pad=0.2", facecolor='white', alpha=0.8))

# Titre
plt.suptitle('Distributions des Features les Plus Importantes', fontsize=16, fontweight='bold', y=0.98)
plt.tight_layout()
plt.subplots_adjust(top=0.92)
plt.show()

# ================================
# FIGURE 2: ANALYSES DISCRIMINANTES
# ================================
fig2 = plt.figure(figsize=(16, 8))
fig2.patch.set_facecolor('#2F2F2F')

colors = ['#4ECDC4', '#FFFFFF', '#FF6B9D']
n_bins = 256
cmap_custom = mcolors.LinearSegmentedColormap.from_list('blue_to_pink', colors, N=n_bins)
```

```
# Subplot 1: Pouvoir Discriminant
ax1 = plt.subplot(1, 2, 1)
ax1.set_facecolor('#3A3A3A')
bars = plt.bar(range(len(discrimination_score)), discrimination_score.values,
               color=['#FF6B9D' if x > discrimination_score.median() else '#4ECDC4'
                      for x in discrimination_score.values])
plt.title('Pouvoir Discriminant des Features', fontsize=14, fontweight='bold')
plt.xticks(range(len(discrimination_score)), discrimination_score.index, rotation=45)
plt.ylabel('Ecart-type entre classes')

# Subplot 2: Matrice de Corrélation avec palette personnalisée bleu vers rose
ax2 = plt.subplot(1, 2, 2)
ax2.set_facecolor('#3A3A3A')
correlation_matrix = df[features_all].corr()
mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))
sns.heatmap(correlation_matrix, mask=mask, annot=True, cmap=cmap_custom, center=0,
           square=True, fmt='.2f', cbar_kws={"shrink": .8}, ax=ax2)
plt.title('Matrice de Correlation Features', fontsize=14, fontweight='bold')

plt.tight_layout()
plt.show()

# ================================
# FIGURE 3: RÉCAPITULATIF COMPLET
# ================================
fig3 = plt.figure(figsize=(16, 10))
fig3.patch.set_facecolor('#2F2F2F')
plt.axis('off')

# Statistiques détaillées
stats_text = "ANALYSE COMPLETE DES FEATURES NUTRITIONNELLES\n" + "="*80 + "\n\n"

# Corrélations importantes
stats_text += "CORRELATIONS IMPORTANTES (>0.7):\n"
stats_text += "-" * 40 + "\n"
corr_pairs = []
for i in range(len(features_all)):
    for j in range(i+1, len(features_all)):
        corr_val = correlation_matrix.iloc[i, j]
        if abs(corr_val) > 0.7:
            corr_pairs.append((features_all[i], features_all[j], corr_val))
```

```
corr_pairs.sort(key=lambda x: abs(x[2]), reverse=True)
for feat1, feat2, corr in corr_pairs:
    stats_text += f"{feat1:<15} <-> {feat2:<15}: {corr:6.3f}\n"

# Recommandations stratégiques
stats_text += "\nRECOMMANDATIONS STRATEGIQUES POUR LE MODELE ML:\n"
stats_text += "-" * 50 + "\n"
top_features = discrimination_score.head(4).index.tolist()
weak_features = discrimination_score.tail(3).index.tolist()

stats_text += "FEATURES A CONSERVER ABSOLUMENT:\n"
for i, feat in enumerate(top_features, 1):
    score = discrimination_score[feat]
    stats_text += f"  {i}. {feat:<20} (score: {score:.4f})\n"

stats_text += "\nFEATURES A EVALUER POUR SUPPRESSION:\n"
for i, feat in enumerate(weak_features, 1):
    score = discrimination_score[feat]
    stats_text += f"  {i}. {feat:<20} (score: {score:.4f})\n"

# Insights nutritionnels
stats_text += "\nINSIGHTS NUTRITIONNELS CLES:\n"
stats_text += "-" * 30 + "\n"
top_feature = discrimination_score.index[0]
bas_val = df[df['calorie_level'] == 'bas'][top_feature].mean()
haut_val = df[df['calorie_level'] == 'haut'][top_feature].mean()

stats_text += f"Feature la plus discriminante: {top_feature}\n"
stats_text += f"- Separe BAS ({bas_val:.3f}) vs HAUT ({haut_val:.3f})\n"
stats_text += f"- Facteur multiplicatif: {haut_val/bas_val:.2f}x\n\n"

stats_text += "Patterns nutritionnels identifiés:\n"
for feature in discrimination_score.head(3).index:
    bas_mean = df[df['calorie_level'] == 'bas'][feature].mean()
    haut_mean = df[df['calorie_level'] == 'haut'][feature].mean()
    if "fat" in feature or "sugar" in feature:
        interpretation = "plus de gras/sucre = plus de calories (logique)"
    elif "vegetable" in feature:
        interpretation = "plus de legumes = moins de calories (inverse)"
    else:
```

```
        interpretation = "pattern nutritionnel complexe"
    stats_text += f"- {feature}: {interpretation}\n"

plt.text(0.05, 0.95, stats_text, transform=plt.gca().transAxes,
         fontsize=11, verticalalignment='top',
         bbox=dict(boxstyle="round,pad=0.5", facecolor="#3A3A3A", alpha=0.9))

plt.tight_layout()
plt.show()
```


Distributions des Features les Plus Importantes
nb_fat
fat_ratio
nb_sugar
sugar_ratio
nb_protein
protein_ratio
nb_vegetable
vegetable_ratio
calorie_level
bas
moyen
haut
1.70
2.49
2.18
0.227
0.263
0.251
1.11
0.95
1.00
0.161
0.109
0.123
0.54
1.06
0.91
0.067
0.107
0.099
1.06
1.47
1.42
0.123
0.134
0.140


Pouvoir Discriminant des Features
Ecart-type entre classes
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
nb_fat
nb_protein
nb_spice
nb_vegetable
nb_grain
nb_sugar
drink_ratio
sugar_ratio
nb_drink
protein_ratio
fat_ratio
vegetable_ratio
grain_ratio
spice_ratio
Matrice de Correlation Features
nb_fat
nb_sugar 0.07
nb_protein 0.31 -0.17
nb_vegetable -0.03 -0.42 0.25
nb_grain 0.30 0.05 0.18 -0.05
nb_drink 0.14 0.66 -0.07 -0.26 0.19
nb_spice -0.01 -0.22 0.19 0.43 0.04 -0.09
fat_ratio 0.75 -0.01 0.08 -0.28 0.09 -0.07 -0.33
sugar_ratio -0.11 0.85 -0.28 -0.47 -0.10 0.50 -0.36 -0.01
protein_ratio 0.18 -0.20 0.84 0.04 0.07 -0.19 -0.05 0.19 -0.25
vegetable_ratio -0.13 -0.48 0.12 0.88 -0.15 -0.39 0.24 -0.24 -0.51 0.04
grain_ratio 0.16 -0.02 0.06 -0.17 0.87 0.06 -0.12 0.14 -0.09 0.07 -0.19
drink_ratio -0.12 0.54 -0.28 -0.43 -0.03 0.78 -0.35 -0.04 0.66 -0.26 -0.48 -0.01
spice_ratio -0.16 -0.30 0.02 0.25 -0.08 -0.25 0.85 -0.32 -0.38 -0.08 0.21 -0.14 -0.38
nb_fat
nb_sugar
nb_protein
nb_vegetable
nb_grain
nb_drink
nb_spice
fat_ratio
sugar_ratio
protein_ratio
vegetable_ratio
grain_ratio
drink_ratio
spice_ratio
0.8
0.6
0.4
0.2
0.0
-0.2
-0.4

```
ANALYSE COMPLETE DES FEATURES NUTRITIONNELLES
================================================================================

CORRELATIONS IMPORTANTES (>0.7):
----------------------------------------
nb_vegetable    <-> vegetable_ratio:  0.880
nb_grain        <-> grain_ratio    :  0.873
nb_sugar        <-> sugar_ratio    :  0.853
nb_spice        <-> spice_ratio    :  0.846
nb_protein      <-> protein_ratio  :  0.837
nb_drink        <-> drink_ratio    :  0.775
nb_fat          <-> fat_ratio      :  0.751

RECOMMANDATIONS STRATEGIQUES POUR LE MODELE ML:
--------------------------------------------------
FEATURES A CONSERVER ABSOLUMENT:
  1. nb_fat               (score: 0.4012)
  2. nb_protein           (score: 0.2673)
  3. nb_spice             (score: 0.2650)
  4. nb_vegetable         (score: 0.2233)

FEATURES A EVALUER POUR SUPPRESSION:
  1. vegetable_ratio      (score: 0.0088)
  2. grain_ratio          (score: 0.0071)
  3. spice_ratio          (score: 0.0040)

INSIGHTS NUTRITIONNELS CLES:
------------------------------
Feature la plus discriminante: nb_fat
- Separe BAS (1.695) vs HAUT (2.492)
- Facteur multiplicatif: 1.47x

Patterns nutritionnels identifiés:
- nb_fat: plus de gras/sucre = plus de calories (logique)
- nb_protein: pattern nutritionnel complexe
- nb_spice: pattern nutritionnel complexe
```

8. Préparation des données d'entrainement (X, y)

In [8]:

```python
# LabelEncoder pour les classes, et en plus ça garantit l'ordre des classes
le = LabelEncoder()
```

```
le.fit(['bas', 'moyen', 'haut'])

y_encoded = le.transform(df['calorie_level'])

tfidf = TfidfVectorizer(
    max_features=500,
    min_df=100,
    max_df=0.95,
    ngram_range=(1, 2),
    stop_words=None
)

# Vectoriser le texte
X_tfidf = tfidf.fit_transform(df['ingredients_cleaned'])

# Features numériques
numeric_features = [
    'n_ingredients',
    'nb_fat', 'nb_sugar', 'nb_drink', 'nb_protein', 'nb_vegetable', 'nb_grain', 'nb_spice',
    'fat_ratio', 'sugar_ratio', 'drink_ratio', 'protein_ratio', 'vegetable_ratio', 'grain_ratio', 'spice_ratio'
]

# Normaliser les features numériques
scaler = StandardScaler()
X_numeric = scaler.fit_transform(df[numeric_features])

# Convertir les features numériques (dense numpy array) en sparse
X_numeric_sparse = csr_matrix(X_numeric)

# Combiner TF-IDF + features numériques + features catégorielles
X_combined = hstack([X_tfidf, X_numeric_sparse])

# Labels de classification encodés
y = y_encoded

print(f"Forme de la matrice TF-IDF: {X_tfidf.shape}")
print(f"Forme des features numériques: {X_numeric.shape}")
print(f"Forme de la matrice hybride: {X_combined.shape}")

# Division des données
X_train, X_test, y_train, y_test = train_test_split(
```

```
    X_combined, y, test_size=0.2, random_state=42
)

print(f"\nTaille du dataset:")
print(f"- Jeu d'entraînement: {X_train.shape[0]:,} échantillons")
print(f"- Jeu de test: {X_test.shape[0]:,} échantillons")
```

```
Forme de la matrice TF-IDF: (228430, 500)
Forme des features numériques: (228430, 15)
Forme de la matrice hybride: (228430, 515)

Taille du dataset:
- Jeu d'entraînement: 182,744 échantillons
- Jeu de test: 45,686 échantillons
```

9. Optimisation des Hyperparamètres avec XGBoost

In [ ]:

```
print("Configuration XGBoost:")

xgb_balanced_base = xgb.XGBClassifier(
    objective='multi:softprob',
    n_jobs=-1,
    random_state=42,
    eval_metric='mlogloss',
    verbosity=1,
    tree_method='hist',
    # Hyperparamètres par défaut
    learning_rate=0.1,
    n_estimators=200,
    max_depth=6,
    subsample=0.8,
    colsample_bytree=0.8,
    min_child_weight=3,
    gamma=0.1,
    reg_alpha=0.2,
    reg_lambda=1.2
)

print(f"Configuration XGBoost:")
print(f"- objective: {xgb_balanced_base.objective}")
```

```python
print(f"- tree_method: {xgb_balanced_base.tree_method}")
print(f"- learning_rate: {xgb_balanced_base.learning_rate}")
print(f"- n_estimators: {xgb_balanced_base.n_estimators}")
print(f"- max_depth: {xgb_balanced_base.max_depth}")

# Validation croisée stratifiée
stratified_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Hyperparamètres à optimiser
param_xgb_balanced = {
    'n_estimators': [100, 200, 300],
    'max_depth': [4, 6, 8],
    'learning_rate': [0.05, 0.1, 0.15],
    'subsample': [0.7, 0.8, 0.9],
    'colsample_bytree': [0.7, 0.8, 0.9],
    'min_child_weight': [1, 3, 5],
    'gamma': [0, 0.1, 0.2],
    'reg_alpha': [0.1, 0.2, 0.3],
    'reg_lambda': [1.0, 1.2, 1.5]
}

search_xgb_balanced = RandomizedSearchCV(
    estimator=xgb_balanced_base,
    param_distributions=param_xgb_balanced,
    n_iter=20,
    cv=stratified_cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1,
    random_state=42
)

print(f"\nOptimisation des hyperparamètres:")
print(f"- CV stratifié: {stratified_cv.n_splits} folds")
print(f"- Scoring: accuracy (simple et efficace)")
print(f"- Paramètres testés: {search_xgb_balanced.n_iter}")
```

```
Configuration XGBoost:
Configuration XGBoost:
- objective: multi:softprob
- tree_method: hist
- learning_rate: 0.1
- n_estimators: 200
- max_depth: 6

Optimisation des hyperparamètres:
- CV stratifié: 5 folds
- Scoring: accuracy (simple et efficace)
- Paramètres testés: 20
```

10. Entraînement avec XGBoost

In [10]:

```
print("\nDémarrage de l'entraînement...")

# Entraînement
search_xgb_balanced.fit(X_train, y_train)

print(f"\nOptimisation terminée!")
print(f"- Meilleurs paramètres: {search_xgb_balanced.best_params_}")
print(f"- Meilleur score accuracy: {search_xgb_balanced.best_score_:.4f}")

# Récupération du meilleur modèle
best_xgb = search_xgb_balanced.best_estimator_
```

```
Démarrage de l'entraînement...
Fitting 5 folds for each of 20 candidates, totalling 100 fits

Optimisation terminée!
- Meilleurs paramètres: {'subsample': 0.7, 'reg_lambda': 1.5, 'reg_alpha': 0.3, 'n_estimators': 300, 'min_child_weight': 1, 'ma
x_depth': 8, 'learning_rate': 0.15, 'gamma': 0.2, 'colsample_bytree': 0.7}
- Meilleur score accuracy: 0.5253
```

11. Récapitulatif sur le meilleur modèle sélectionné

In [11]:
```
print(f"\nPARAMÈTRES FINAUX:")
print(f"- objective: {best_xgb.objective}")
print(f"- min_child_weight: {best_xgb.min_child_weight}")
print(f"- gamma: {best_xgb.gamma}")
print(f"- reg_alpha: {best_xgb.reg_alpha}")
print(f"- reg_lambda: {best_xgb.reg_lambda}")
print(f"- tree_method: {best_xgb.tree_method}")
print(f"- n_estimators: {best_xgb.n_estimators}")
print(f"- max_depth: {best_xgb.max_depth}")
print(f"- learning_rate: {best_xgb.learning_rate}")

print(f"\nRÉCAPITULATIF DE L'APPROCHE:")
print(f"- Données: originales")
print(f"- Équilibrage: sample weights calculés")
print(f"- Métrique: accuracy")
print(f"- Validation: StratifiedKFold (5 folds)")
print(f"- Optimisation: RandomizedSearchCV (20 itérations)")
print(f"- Performance: {search_xgb_balanced.best_score_:.4f} accuracy")
```

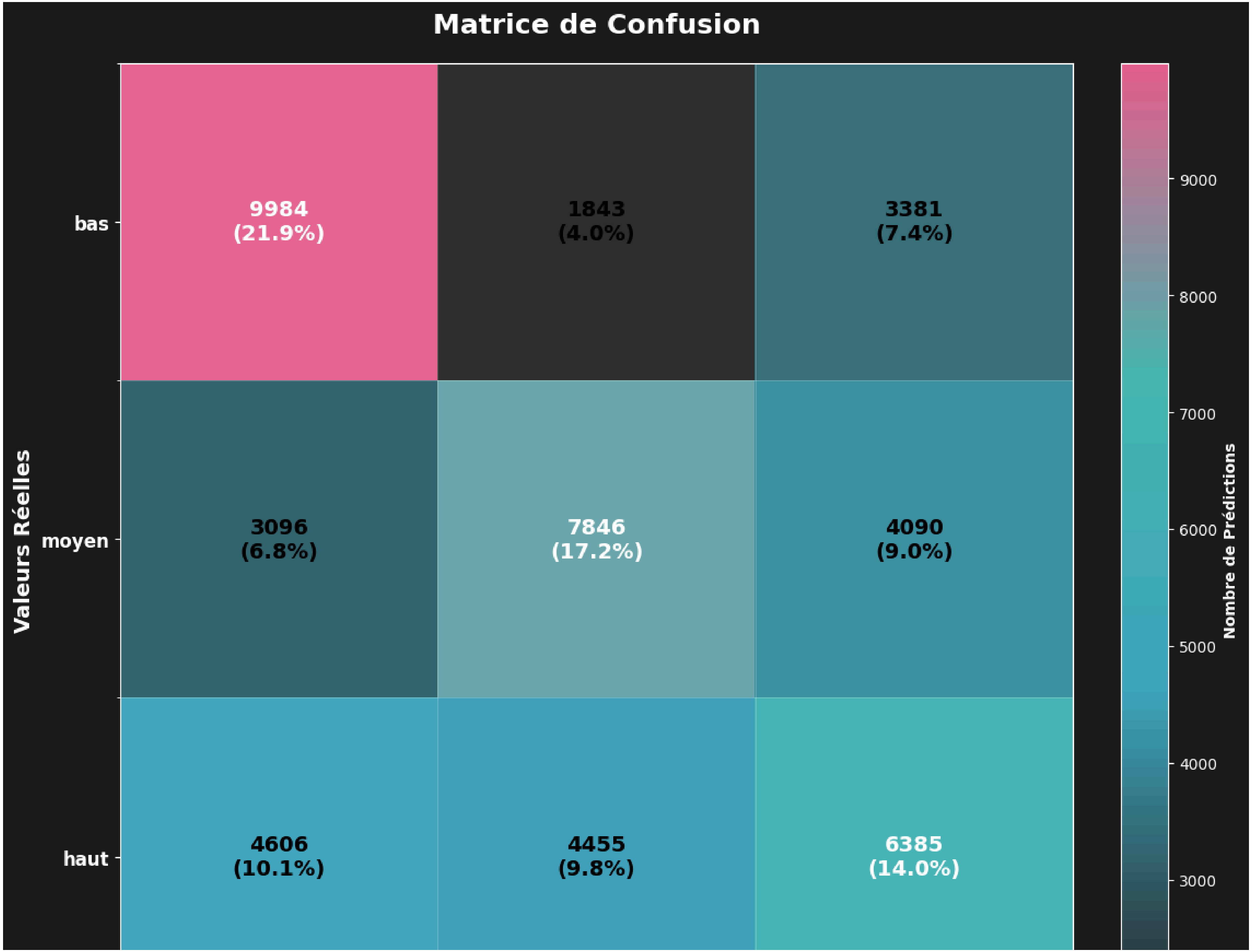
```
PARAMÈTRES FINAUX:
- objective: multi:softprob
- min_child_weight: 1
- gamma: 0.2
- reg_alpha: 0.3
- reg_lambda: 1.5
- tree_method: hist
- n_estimators: 300
- max_depth: 8
- learning_rate: 0.15

RÉCAPITULATIF DE L'APPROCHE:
- Données: originales
- Équilibrage: sample weights calculés
- Métrique: accuracy
- Validation: StratifiedKFold (5 folds)
- Optimisation: RandomizedSearchCV (20 itérations)
- Performance: 0.5253 accuracy
```

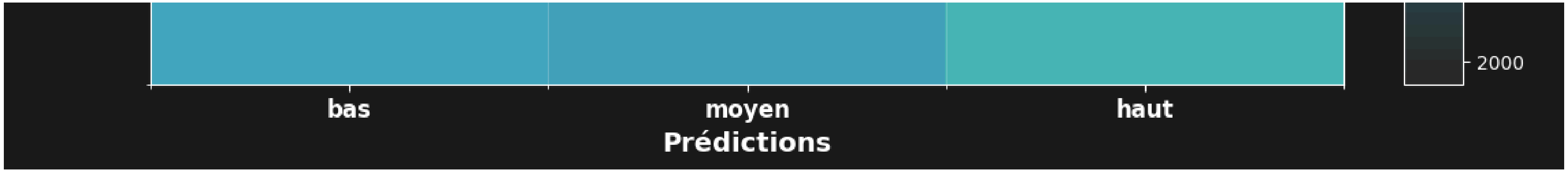
12. Évaluation complète du meilleur modèle sélectionné

In [ ]:
```
# Prédictions
y_pred_train = best_xgb.predict(X_train)
y_pred_test = best_xgb.predict(X_test)

# Scores d'accuracy
train_accuracy = accuracy_score(y_train, y_pred_train)
test_accuracy = accuracy_score(y_test, y_pred_test)

print(f"Accuracy d'entraînement: {train_accuracy:.4f}")
print(f"Accuracy de test: {test_accuracy:.4f}")

# Rapport de classification
print("\nRapport de classification (jeu de test):")
print(classification_report(y_test, y_pred_test, target_names=['bas', 'moyen', 'haut']))

# Matrice de confusion
fig, ax = plt.subplots(figsize=(12, 10))
fig.patch.set_facecolor('#1a1a1a')
ax.set_facecolor('#2d2d2d')

cm = confusion_matrix(y_test, y_pred_test)

# Créer un heatmap
import matplotlib.colors as mcolors

colors = ['#2d2d2d', '#45B7D1', '#4ECDC4', '#FF6B9D']
n_bins = 100
cmap = mcolors.LinearSegmentedColormap.from_list('custom', colors, N=n_bins)

im = ax.imshow(cm, interpolation='nearest', cmap=cmap, alpha=0.9)

for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        text_color = 'white' if cm[i, j] > cm.max() / 2 else 'black'
        ax.text(j, i, f'{cm[i, j]}\n({cm[i, j]/cm.sum()*100:.1f}%)',
                ha='center', va='center', fontweight='bold',
                color=text_color, fontsize=14)

class_names = ['bas', 'moyen', 'haut']
ax.set_xticks(range(len(class_names)))
```

```python
ax.set_yticks(range(len(class_names)))
ax.set_xticklabels(class_names, fontsize=12, color='white', fontweight='bold')
ax.set_yticklabels(class_names, fontsize=12, color='white', fontweight='bold')

# Titres et labels
ax.set_title('Matrice de Confusion', fontsize=18, fontweight='bold',
             color='white', pad=20)
ax.set_xlabel('Prédictions', fontsize=14, fontweight='bold', color='white')
ax.set_ylabel('Valeurs Réelles', fontsize=14, fontweight='bold', color='white')

cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
cbar.ax.yaxis.set_tick_params(color='white')
cbar.ax.tick_params(labelcolor='white')
cbar.set_label('Nombre de Prédictions', color='white', fontweight='bold')

ax.set_xticks(np.arange(len(class_names) + 1) - 0.5, minor=True)
ax.set_yticks(np.arange(len(class_names) + 1) - 0.5, minor=True)
ax.grid(which='minor', color='white', linestyle='-', linewidth=0.5, alpha=0.3)

plt.tight_layout()
plt.show()

# Validation croisée
cv_scores = cross_val_score(best_xgb, X_train, y_train, cv=5, scoring='accuracy')
print(f"\nScores de validation croisée: {cv_scores}")
print(f"Score moyen: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4f})")
```

```
Accuracy d'entraînement: 0.6501
Accuracy de test: 0.5300

Rapport de classification (jeu de test):
              precision    recall  f1-score   support

         bas       0.56      0.66      0.61     15208
       moyen       0.55      0.52      0.54     15032
        haut       0.46      0.41      0.44     15446

    accuracy                           0.53     45686
   macro avg       0.53      0.53      0.53     45686
weighted avg       0.53      0.53      0.53     45686
```


Matrice de Confusion
Valeurs Réelles
bas
moyen
haut
9984
(21.9%)
1843
(4.0%)
3381
(7.4%)
3096
(6.8%)
7846
(17.2%)
4090
(9.0%)
4606
(10.1%)
4455
(9.8%)
6385
(14.0%)
Nombre de Prédictions
9000
8000
7000
6000
5000
4000
3000




```
Scores de validation croisée: [0.52450135 0.52332485 0.52302389 0.52217571 0.52604794]
Score moyen: 0.5238 (+/- 0.0027)
```

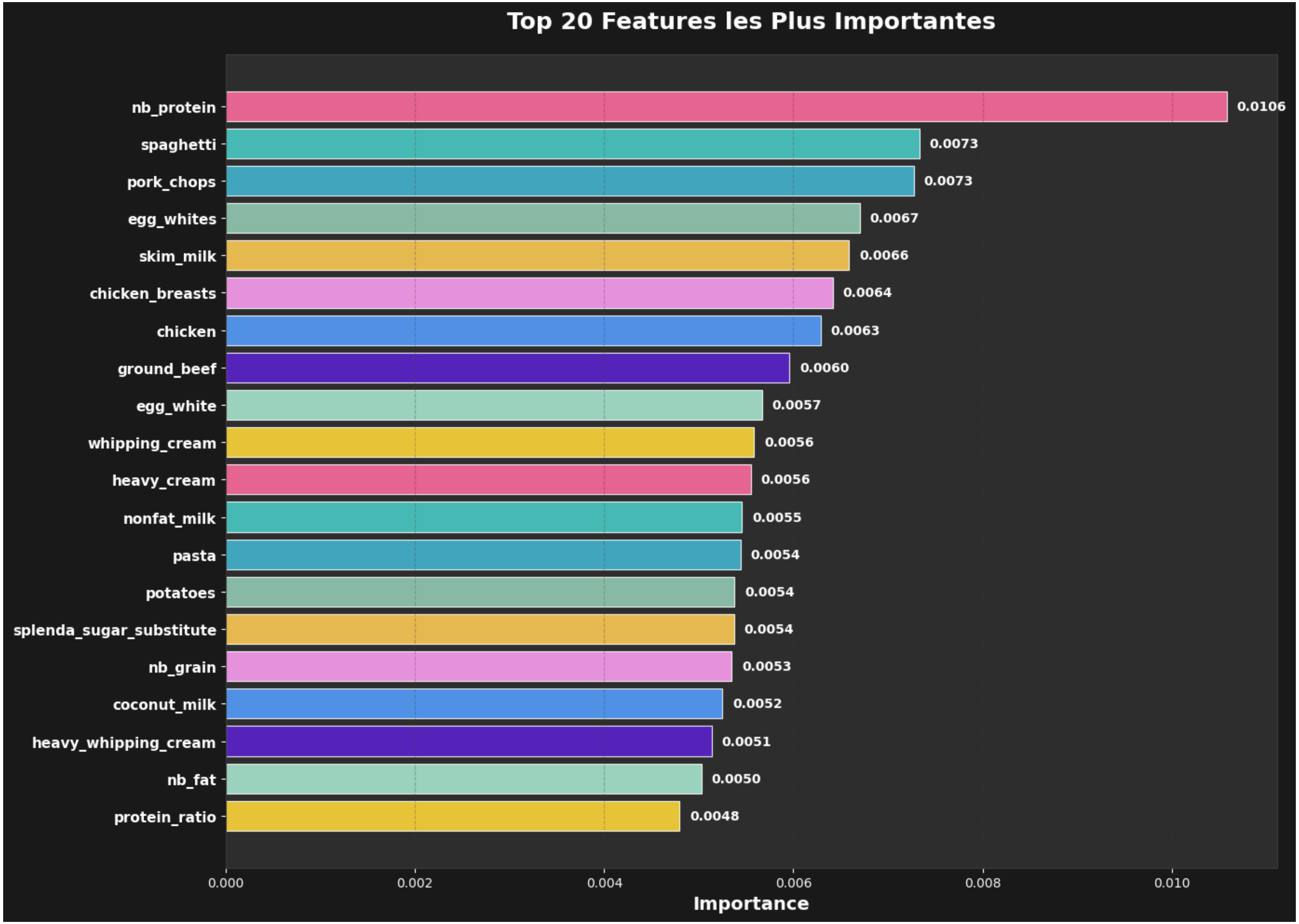
13. Importance des features

In [ ]:
```python
# Récupération des noms de features
def get_all_feature_names(tfidf_vectorizer, numeric_features):
    """Récupère tous les noms de features dans le bon ordre"""
    # 1. Features TF-IDF
    tfidf_names = list(tfidf_vectorizer.get_feature_names_out())

    # 2. Features numériques
    numeric_names = numeric_features.copy()

    # Combiner dans le même ordre que lors de la création de X_combined
    all_feature_names = tfidf_names + numeric_names

    return all_feature_names

numeric_features_list = [
    'n_ingredients',
    'nb_fat', 'nb_sugar', 'nb_drink', 'nb_protein', 'nb_vegetable', 'nb_grain', 'nb_spice',
    'fat_ratio', 'sugar_ratio', 'drink_ratio', 'protein_ratio', 'vegetable_ratio', 'grain_ratio', 'spice_ratio'
]

# Obtenir TOUS les noms de features
all_feature_names = get_all_feature_names(tfidf, numeric_features_list)
feature_importance = best_xgb.feature_importances_

print(f"Nombre de noms de features: {len(all_feature_names)}")
print(f"Nombre d'importances: {len(feature_importance)}")
print(f"Match: {len(all_feature_names) == len(feature_importance)}")
```

```python
# Créer le DataFrame des importances
importance_df = pd.DataFrame({
    'feature': all_feature_names,
    'importance': feature_importance
}).sort_values('importance', ascending=False)

# Graphique
fig, ax = plt.subplots(figsize=(14, 10))
fig.patch.set_facecolor('#1a1a1a')
ax.set_facecolor('#2d2d2d')

top_features = importance_df.head(20)

beautiful_colors = ['#FF6B9D', '#4ECDC4', '#45B7D1', '#96CEB4', '#FECA57',
                   '#FF9FF3', '#54A0FF', '#5F27CD', '#A8E6CF', '#FFD93D']

colors_bars = [beautiful_colors[i % len(beautiful_colors)] for i in range(len(top_features))]
bars = ax.barh(range(len(top_features)), top_features['importance'],
               color=colors_bars, alpha=0.9,
               edgecolor='white', linewidth=0.8)

# Configuration des axes
ax.set_yticks(range(len(top_features)))
ax.set_yticklabels(top_features['feature'], fontsize=11, color='white', fontweight='bold')
ax.set_xlabel('Importance', fontweight='bold', color='white', fontsize=14)
ax.set_title('Top 20 Features les Plus Importantes',
             fontweight='bold', fontsize=18, color='white', pad=20)

for i, (bar, importance) in enumerate(zip(bars, top_features['importance'])):
    ax.text(bar.get_width() + max(top_features['importance'])*0.01,
            bar.get_y() + bar.get_height()/2,
            f'{importance:.4f}',
            ha='left', va='center', fontweight='bold',
            color='white', fontsize=10)

ax.invert_yaxis()
ax.tick_params(colors='white')
ax.grid(True, alpha=0.3, color='#404040', linestyle='--', axis='x')

for spine in ax.spines.values():
    spine.set_color('#404040')
```

```
plt.tight_layout()
plt.show()

print("Top 10 des features les plus importantes:")
print(importance_df.head(10))

# Analyse par type de feature
print("\n" + "="*60)
print("ANALYSE PAR TYPE DE FEATURE")
print("="*60)

# Top features numériques
numeric_importance = importance_df[importance_df['feature'].isin(numeric_features_list)]
print("\nTop 5 Features Numériques:")
print(numeric_importance.head())

# Top features TF-IDF
tfidf_importance = importance_df[~importance_df['feature'].isin(numeric_features_list)]
print("\nTop 5 Features TF-IDF (ingrédients):")
print(tfidf_importance.head())
```

```
Nombre de noms de features: 515
Nombre d'importances: 515
Match: True
```


Top 20 Features les Plus Importantes
nb_protein 0.0106
spaghetti 0.0073
pork_chops 0.0073
egg_whites 0.0067
skim_milk 0.0066
chicken_breasts 0.0064
chicken 0.0063
ground_beef 0.0060
egg_white 0.0057
whipping_cream 0.0056
heavy_cream 0.0056
nonfat_milk 0.0055
pasta 0.0054
potatoes 0.0054
splenda_sugar_substitute 0.0054
nb_grain 0.0053
coconut_milk 0.0052
heavy_whipping_cream 0.0051
nb_fat 0.0050
protein_ratio 0.0048
0.000
0.002
0.004
0.006
0.008
0.010
Importance

```
Top 10 des features les plus importantes:
             feature  importance
504       nb_protein    0.010585
436        spaghetti    0.007332
357       pork_chops    0.007272
161       egg_whites    0.006705
428        skim_milk    0.006589
94   chicken_breasts    0.006419
93           chicken    0.006292
224      ground_beef    0.005959
160        egg_white    0.005667
485   whipping_cream    0.005587

============================================================
ANALYSE PAR TYPE DE FEATURE
============================================================

Top 5 Features Numériques:
           feature  importance
504     nb_protein    0.010585
506       nb_grain    0.005348
501         nb_fat    0.005031
511  protein_ratio    0.004796
500  n_ingredients    0.003447

Top 5 Features TF-IDF (ingrédients):
             feature  importance
436        spaghetti    0.007332
357       pork_chops    0.007272
161       egg_whites    0.006705
428        skim_milk    0.006589
94   chicken_breasts    0.006419
```

14. Analyse SHAP pour l'explicabilité

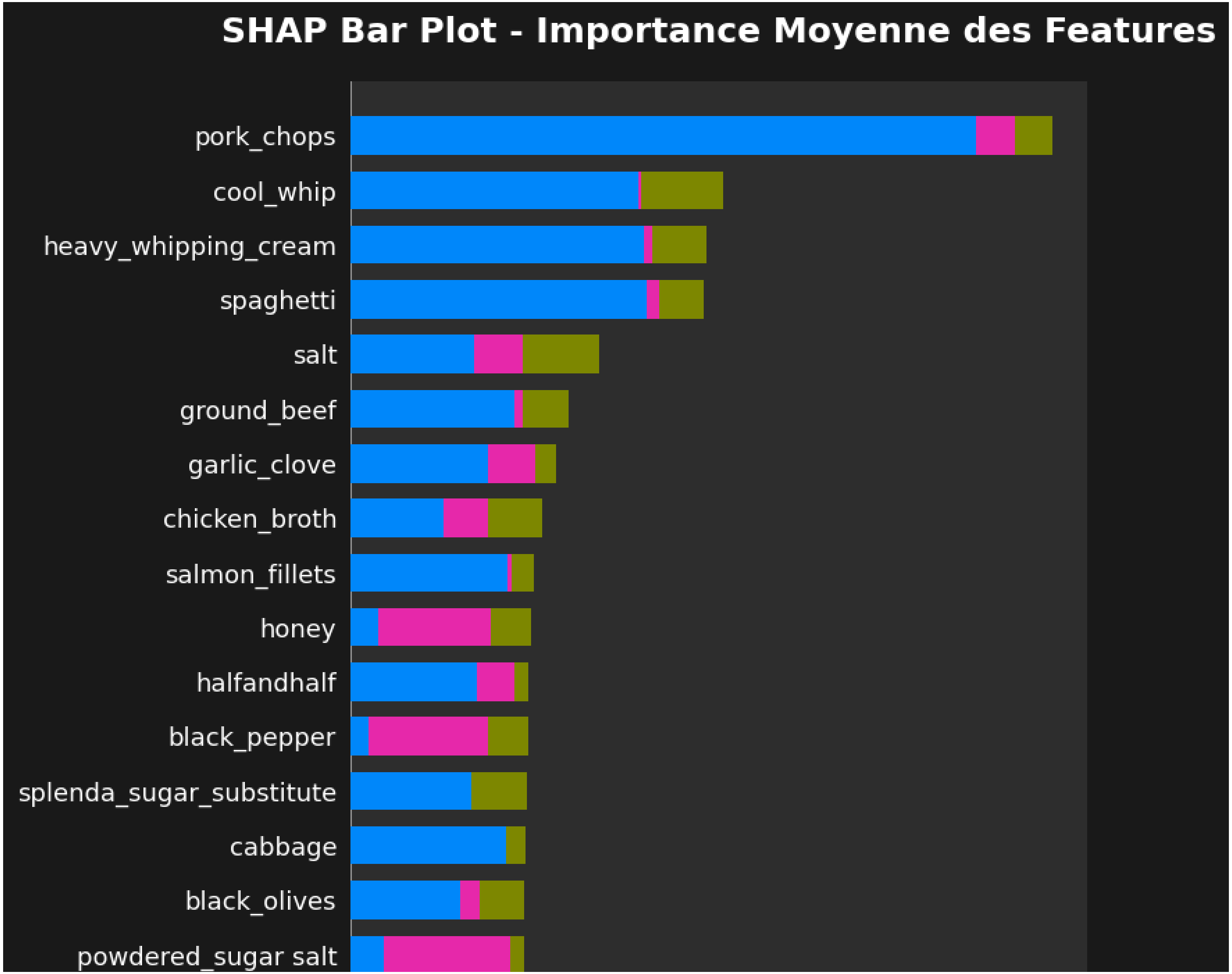
In [ ]:
```
print("Initialisation de l'explainer SHAP...")
explainer = shap.TreeExplainer(best_xgb)

# Calculer les valeurs SHAP sur un échantillon
```

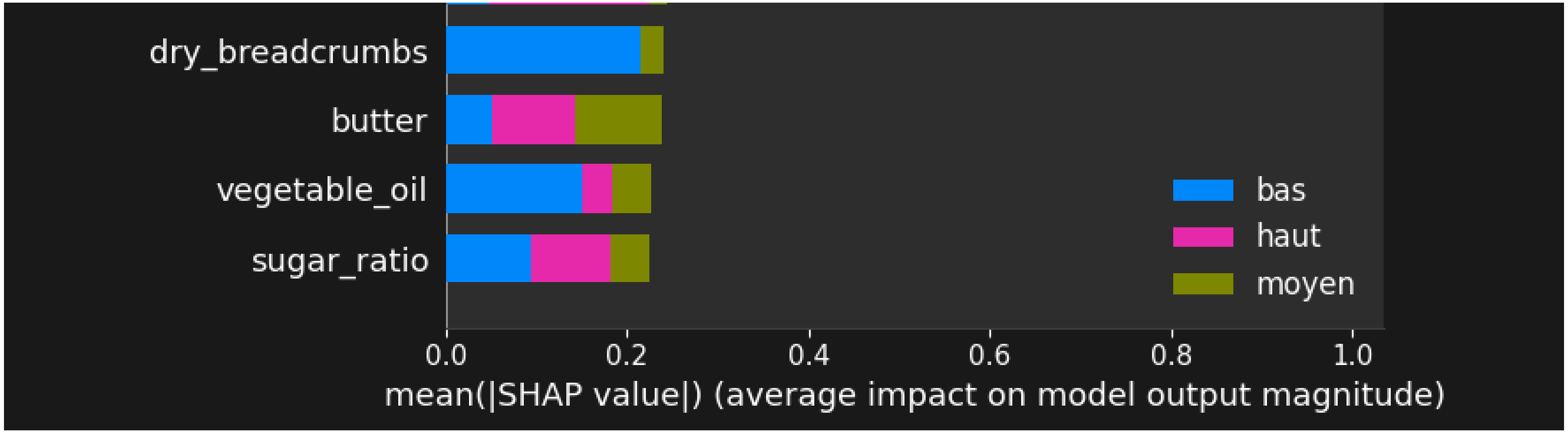
```
sample_size = min(100, X_test.shape[0])
X_test_sample = X_test[:sample_size].toarray().astype(np.float64)
y_test_sample = y_test[:sample_size]

print(f"Calcul des valeurs SHAP pour {sample_size} échantillons...")
shap_values = explainer.shap_values(X_test_sample)

print("Analyse SHAP terminée!")

# Vérification des dimensions
print(f"Forme shap_values: {np.array(shap_values).shape}")
print(f"Forme X_test_sample: {X_test_sample.shape}")
print(f"Nombre de feature names: {len(all_feature_names)}")

# Bar plot SHAP
fig, ax = plt.subplots(figsize=(14, 8))
fig.patch.set_facecolor('#1a1a1a')

shap.summary_plot(shap_values, X_test_sample,
                  feature_names=all_feature_names,
                  plot_type="bar",
                  class_names=['bas', 'moyen', 'haut'],
                  show=False)

ax = plt.gca()
ax.set_facecolor('#2d2d2d')
ax.set_title('SHAP Bar Plot - Importance Moyenne des Features',
             fontweight='bold', fontsize=18, color='white', pad=20)

ax.tick_params(colors='white')
ax.xaxis.label.set_color('white')
ax.yaxis.label.set_color('white')

for spine in ax.spines.values():
    spine.set_color('#404040')

plt.tight_layout()
plt.show()

print("\n" + "="*60)
print("ANALYSE SHAP PAR TYPE DE FEATURE")
```

```python
print("="*60)

print(f"DEBUG: Forme de shap_values: {np.array(shap_values).shape}")

if isinstance(shap_values, list):
    # Cas 1: shap_values est une liste d'arrays (rare)
    mean_shap_importance = np.mean([np.abs(sv).mean(axis=0) for sv in shap_values], axis=0)
else:
    # Cas 2: shap_values est un array 3D (100, 3016, 3) ← notre cas
    if len(shap_values.shape) == 3:
        # Prendre la moyenne absolue sur les échantillons (axis=0) et les classes (axis=2)
        mean_shap_importance = np.abs(shap_values).mean(axis=0).mean(axis=1)
    else:
        # Cas classique 2D
        mean_shap_importance = np.abs(shap_values).mean(axis=0)

print(f"DEBUG: Forme de mean_shap_importance: {mean_shap_importance.shape}")

# Vérification avant création du DataFrame
if mean_shap_importance.ndim != 1:
    print(f"ERREUR: mean_shap_importance doit être 1D, mais a {mean_shap_importance.ndim} dimensions")
    print(f"Forme actuelle: {mean_shap_importance.shape}")
    # Forcer à 1D si nécessaire
    mean_shap_importance = mean_shap_importance.flatten()

# Créer DataFrame SHAP
shap_importance_df = pd.DataFrame({
    'feature': all_feature_names,
    'shap_importance': mean_shap_importance
}).sort_values('shap_importance', ascending=False)

print("Top 10 Features selon SHAP:")
print(shap_importance_df.head(10))

# Comparaison XGBoost vs SHAP importance
comparison_df = importance_df.merge(shap_importance_df, on='feature', how='inner')
comparison_df['rank_xgb'] = comparison_df['importance'].rank(ascending=False)
comparison_df['rank_shap'] = comparison_df['shap_importance'].rank(ascending=False)
comparison_df['rank_diff'] = abs(comparison_df['rank_xgb'] - comparison_df['rank_shap'])
```

```python
print(f"\nComparaison XGBoost vs SHAP (Top 10):")
print(comparison_df.head(10)[['feature', 'importance', 'shap_importance', 'rank_xgb', 'rank_shap']])
```

```
Initialisation de l'explainer SHAP...
Calcul des valeurs SHAP pour 100 échantillons...
Analyse SHAP terminée!
Forme shap_values: (100, 515, 3)
Forme X_test_sample: (100, 515)
Nombre de feature names: 515
```


SHAP Bar Plot - Importance Moyenne des Features
pork_chops
cool_whip
heavy_whipping_cream
spaghetti
salt
ground_beef
garlic_clove
chicken_broth
salmon_fillets
honey
halfandhalf
black_pepper
splenda_sugar_substitute
cabbage
black_olives
powdered_sugar salt


dry_breadcrumbs
butter
vegetable_oil
sugar_ratio
bas
haut
moyen
0.0
0.2
0.4
0.6
0.8
1.0
mean(|SHAP value|) (average impact on model output magnitude)

```
============================================================
ANALYSE SHAP PAR TYPE DE FEATURE
============================================================
DEBUG: Forme de shap_values: (100, 515, 3)
DEBUG: Forme de mean_shap_importance: (515,)
Top 10 Features selon SHAP:
                  feature  shap_importance
357            pork_chops         0.328539
121             cool_whip         0.174171
243  heavy_whipping_cream         0.166342
436             spaghetti         0.165064
392                  salt         0.116480
224           ground_beef         0.101990
206          garlic_clove         0.096341
95          chicken_broth         0.089980
390        salmon_fillets         0.085715
245                 honey         0.084388

Comparaison XGBoost vs SHAP (Top 10):
           feature  importance  shap_importance  rank_xgb  rank_shap
0       nb_protein    0.010585         0.033837       1.0      170.0
1        spaghetti    0.007332         0.165064       2.0        4.0
2       pork_chops    0.007272         0.328539       3.0        1.0
3       egg_whites    0.006705         0.035612       4.0      156.0
4        skim_milk    0.006589         0.022001       5.0      294.0
5  chicken_breasts    0.006419         0.038353       6.0      137.0
6          chicken    0.006292         0.054005       7.0       67.0
7      ground_beef    0.005959         0.101990       8.0        6.0
8        egg_white    0.005667         0.036858       9.0      149.0
9   whipping_cream    0.005587         0.052089      10.0       74.0
```

15. Prédictions

15.1. Fonction de prédiction

In [15]:
```python
def predict_calorie_level(ingredients_text):
    """
    Prédit le niveau calorique avec TOUTES les nouvelles features
```

```
    Args:
        ingredients_text (str): Liste des ingrédients

    Returns:
        tuple: (prédiction, probabilités)
    """

    # === ÉTAPE 1: PREPROCESSING ===
    ingredients_sorted = sort_ingredients(ingredients_text)
    ingredients_cleaned = clean_text(ingredients_sorted)

    print(f"Original: {ingredients_text}")
    print(f"Cleaned: {ingredients_cleaned}")

    # === ÉTAPE 2: TF-IDF ===
    text_vectorized = tfidf.transform([ingredients_cleaned])

    # === ÉTAPE 3: FONCTIONS DE COMPTAGE GÉNÉRIQUES ===
    def count_ingredients_single(ingredients_cleaned, ingredient_list):
        """Fonction générique pour compter une catégorie d'ingrédients"""
        ingredients_list = ingredients_cleaned.lower().split()
        count = 0

        for ingredient in ingredients_list:
            if ingredient in ingredient_list:
                count += 1
            else:
                for target_ing in ingredient_list:
                    if target_ing in ingredient:
                        count += 1
                        break
        return count

    # === ÉTAPE 4: CALCUL DES FEATURES NUMÉRIQUES COMPLÈTES ===
    n_ingredients = len(ingredients_cleaned.split())

    # Compteurs par catégorie
    nb_fat = count_ingredients_single(ingredients_cleaned, fat_ingredients)
    nb_sugar = count_ingredients_single(ingredients_cleaned, sugar_ingredients)
    nb_drink = count_ingredients_single(ingredients_cleaned, drink_ingredients)
    nb_protein = count_ingredients_single(ingredients_cleaned, protein_ingredients)
```

```
    nb_vegetable = count_ingredients_single(ingredients_cleaned, vegetable_ingredients)
    nb_grain = count_ingredients_single(ingredients_cleaned, grain_ingredients)
    nb_spice = count_ingredients_single(ingredients_cleaned, spice_ingredients)

    # Ratios avec epsilon
    epsilon = 1e-6
    fat_ratio = nb_fat / (n_ingredients + epsilon)
    sugar_ratio = nb_sugar / (n_ingredients + epsilon)
    drink_ratio = nb_drink / (n_ingredients + epsilon)
    protein_ratio = nb_protein / (n_ingredients + epsilon)
    vegetable_ratio = nb_vegetable / (n_ingredients + epsilon)
    grain_ratio = nb_grain / (n_ingredients + epsilon)
    spice_ratio = nb_spice / (n_ingredients + epsilon)

    print(f"Features numériques calculées:")
    print(f"- n_ingredients: {n_ingredients}")
    print(f"- Compteurs: fat={nb_fat}, sugar={nb_sugar}, protein={nb_protein}, vegetable={nb_vegetable}")
    print(f"- Compteurs: grain={nb_grain}, drink={nb_drink}, spice={nb_spice}")
    print(f"- Ratios: fat={fat_ratio:.3f}, sugar={sugar_ratio:.3f}, protein={protein_ratio:.3f}")

    # Vecteur numérique
    numeric_values = np.array([[
        n_ingredients, nb_fat, nb_sugar, nb_drink, nb_protein,
        nb_vegetable, nb_grain, nb_spice,
        fat_ratio, sugar_ratio, drink_ratio, protein_ratio,
        vegetable_ratio, grain_ratio, spice_ratio
    ]])

    # Normalisation
    numeric_normalized = scaler.transform(numeric_values)
    numeric_sparse = csr_matrix(numeric_normalized)

    # Combinaison des features
    X_combined_prediction = hstack([text_vectorized, numeric_sparse])

    print(f"\nDimensions finales:")
    print(f"- TF-IDF: {text_vectorized.shape}")
    print(f"- Numériques: {numeric_sparse.shape}")
    print(f"- Combinées: {X_combined_prediction.shape}")
    print(f"- Modèle attend: {X_train.shape[1]} features")
```

```
    # Vérification des dimensions
    if X_combined_prediction.shape[1] != X_train.shape[1]:
        print(f"ERREUR: Mismatch de dimensions!")
        print(f"Attendu: {X_train.shape[1]}, Reçu: {X_combined_prediction.shape[1]}")
        return None, None

    # Prédiction
    prediction_encoded = best_xgb.predict(X_combined_prediction)[0]
    probabilities = best_xgb.predict_proba(X_combined_prediction)[0]

    # Décoder
    prediction = le.inverse_transform([prediction_encoded])[0]
    class_names = le.classes_
    prob_dict = dict(zip(class_names, probabilities))

    print(f"\nRésultat:")
    print(f"- Prédiction: {prediction}")
    print(f"- Confiance: {max(prob_dict.values()):.1%}")
    for class_name, prob in prob_dict.items():
        print(f"- {class_name}: {prob:.1%}")

    return prediction, prob_dict

# test simple
test_ingredients = "['chicken', 'olive_oil', 'garlic', 'tomatoes', 'basil']"
prediction, probabilities = predict_calorie_level(test_ingredients)
```

```
Original: ['chicken', 'olive_oil', 'garlic', 'tomatoes', 'basil']
Cleaned: basil chicken garlic olive_oil tomatoes
Features numériques calculées:
- n_ingredients: 5
- Compteurs: fat=1, sugar=0, protein=1, vegetable=2
- Compteurs: grain=0, drink=0, spice=1
- Ratios: fat=0.200, sugar=0.000, protein=0.200

Dimensions finales:
- TF-IDF: (1, 500)
- Numériques: (1, 15)
- Combinées: (1, 515)
- Modèle attend: 515 features

Résultat:
- Prédiction: haut
- Confiance: 39.7%
- bas: 29.3%
- haut: 39.7%
- moyen: 31.1%
```

15.2. Visualisation d'une prédiction

In [ ]:
```python
def visualize_prediction(ingredients_text):
    """
    Visualise une prédiction avec le thème harmonisé - VERSION CORRIGÉE

    Args:
        ingredients_text (str): Liste des ingrédients
    """
    # Prédiction
    prediction, prob_dict = predict_calorie_level(ingredients_text)

    # Viz
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(18, 12))
    fig.patch.set_facecolor('#1a1a1a')

    # Graphique des probabilités (camembert)
    ax1.set_facecolor('#2d2d2d')
```

```
    category_colors = {'bas': '#96CEB4', 'moyen': '#4ECDC4', 'haut': '#FF6B9D'}
    colors = [category_colors[cat] for cat in prob_dict.keys()]

    explode = [0.1 if cat == prediction else 0 for cat in prob_dict.keys()]

    wedges, texts, autotexts = ax1.pie(prob_dict.values(),
                                       labels=[f'{cat.upper()}\n{prob:.1%}' for cat, prob in prob_dict.items()],
                                       colors=colors, explode=explode, autopct='',
                                       shadow=True, startangle=90,
                                       textprops={'fontsize': 12, 'color': 'white', 'fontweight': 'bold'})

    ax1.set_title(f'Prédiction: {prediction.upper()}',
                 fontweight='bold', fontsize=16, color='white', pad=20)

    # Graphique en barres des probabilités
    ax2.set_facecolor('#2d2d2d')

    categories = list(prob_dict.keys())
    probabilities = list(prob_dict.values())
    colors_bars = [category_colors[cat] for cat in categories]

    bars = ax2.bar(categories, probabilities, color=colors_bars, alpha=0.9,
                   edgecolor='white', linewidth=1.5)

    for i, (bar, cat) in enumerate(zip(bars, categories)):
        if cat == prediction:
            bar.set_edgecolor('#FFD93D')
            bar.set_linewidth(3)

        ax2.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.01,
                f'{probabilities[i]:.1%}', ha='center', va='bottom',
                fontweight='bold', color='white', fontsize=12)

    ax2.set_title('Probabilités par Catégorie',
                 fontweight='bold', fontsize=16, color='white', pad=20)
    ax2.set_ylabel('Probabilité', fontweight='bold', color='white')
    ax2.tick_params(colors='white')
    ax2.grid(True, alpha=0.3, color='#404040', linestyle='--')

    for spine in ax2.spines.values():
        spine.set_color('#404040')
```

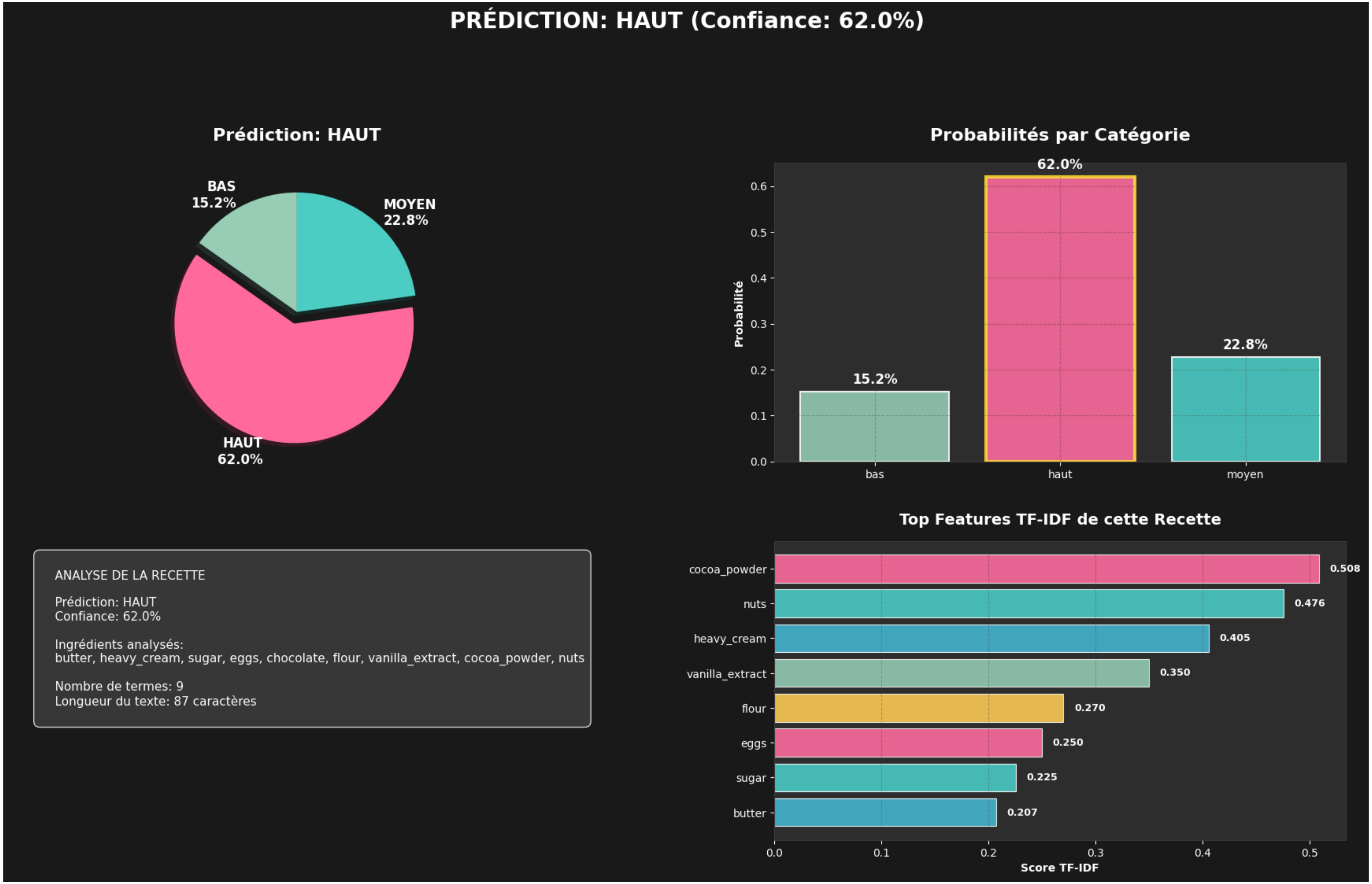
```
    # Texte des ingrédients
    ax3.set_facecolor('#2d2d2d')
    ax3.axis('off')

    ingredients_clean = clean_text(ingredients_text)
    ingredients_words = ingredients_clean.split()

    ingredients_display = ', '.join(ingredients_words[:10])
    if len(ingredients_words) > 10:
        ingredients_display += f"... (+{len(ingredients_words) - 10} mots)"

    info_text = f"""
    ANALYSE DE LA RECETTE

    Prédiction: {prediction.upper()}
    Confiance: {max(prob_dict.values()):.1%}

    Ingrédients analysés:
    {ingredients_display}

    Nombre de termes: {len(ingredients_words)}
    Longueur du texte: {len(ingredients_text)} caractères
    """

    ax3.text(0.05, 0.95, info_text, transform=ax3.transAxes,
             fontsize=11, color='white', va='top', ha='left',
             bbox=dict(boxstyle="round,pad=0.5", facecolor='#404040', alpha=0.8))

    # Features TF-IDF de cette prédiction
    ax4.set_facecolor('#2d2d2d')

    text_vectorized = tfidf.transform([ingredients_clean])

    if text_vectorized.nnz > 0:
        feature_indices = text_vectorized.nonzero()[1]
        feature_scores = text_vectorized.data

        # Utiliser les noms TF-IDF valides
        tfidf_feature_names = tfidf.get_feature_names_out()
        prediction_features = pd.DataFrame({
```

```
            'feature': [tfidf_feature_names[i] for i in feature_indices],
            'score': feature_scores
        }).sort_values('score', ascending=False).head(10)

        beautiful_colors = ['#FF6B9D', '#4ECDC4', '#45B7D1', '#96CEB4', '#FECA57']
        colors_features = [beautiful_colors[i % len(beautiful_colors)] for i in range(len(prediction_features))]
        bars = ax4.barh(range(len(prediction_features)), prediction_features['score'],
                       color=colors_features, alpha=0.9,
                       edgecolor='white', linewidth=0.8)

        ax4.set_yticks(range(len(prediction_features)))
        ax4.set_yticklabels(prediction_features['feature'], fontsize=10, color='white')
        ax4.set_xlabel('Score TF-IDF', fontweight='bold', color='white')
        ax4.set_title('Top Features TF-IDF de cette Recette',
                     fontweight='bold', fontsize=14, color='white', pad=15)

        for i, (bar, score) in enumerate(zip(bars, prediction_features['score'])):
            ax4.text(bar.get_width() + max(prediction_features['score'])*0.02,
                    bar.get_y() + bar.get_height()/2,
                    f'{score:.3f}',
                    ha='left', va='center', fontweight='bold',
                    color='white', fontsize=9)

        ax4.invert_yaxis()
        ax4.tick_params(colors='white')
        ax4.grid(True, alpha=0.3, color='#404040', linestyle='--', axis='x')
    else:
        ax4.text(0.5, 0.5, 'Aucune feature trouvée', ha='center', va='center',
                transform=ax4.transAxes, color='white', fontsize=14)

    for spine in ax4.spines.values():
        spine.set_color('#404040')

    fig.suptitle(f'PRÉDICTION: {prediction.upper()} (Confiance: {max(prob_dict.values()):.1%})',
                fontsize=20, fontweight='bold', color='white', y=0.98)

    plt.tight_layout(pad=3.0, rect=[0, 0.03, 1, 0.95])
    plt.show()

    return prediction, prob_dict
```

15.3. Test dessert riche

In [17]:
```
# Exemple avec un dessert riche
dessert_ingredients = "butter, heavy cream, sugar, eggs, chocolate, flour, vanilla extract, cocoa powder, nuts"
print(f"\nExemple 1 - Dessert riche:")
print(f"Ingrédients: {dessert_ingredients}")
pred1, prob1 = visualize_prediction(dessert_ingredients)
```

```
Exemple 1 - Dessert riche:
Ingrédients: butter, heavy cream, sugar, eggs, chocolate, flour, vanilla extract, cocoa powder, nuts
Original: butter, heavy cream, sugar, eggs, chocolate, flour, vanilla extract, cocoa powder, nuts
Cleaned: butter heavy_cream sugar eggs chocolate flour vanilla_extract cocoa_powder nuts
Features numériques calculées:
- n_ingredients: 9
- Compteurs: fat=4, sugar=3, protein=1, vegetable=0
- Compteurs: grain=1, drink=3, spice=0
- Ratios: fat=0.444, sugar=0.333, protein=0.111

Dimensions finales:
- TF-IDF: (1, 500)
- Numériques: (1, 15)
- Combinées: (1, 515)
- Modèle attend: 515 features

Résultat:
- Prédiction: haut
- Confiance: 62.0%
- bas: 15.2%
- haut: 62.0%
- moyen: 22.8%
```

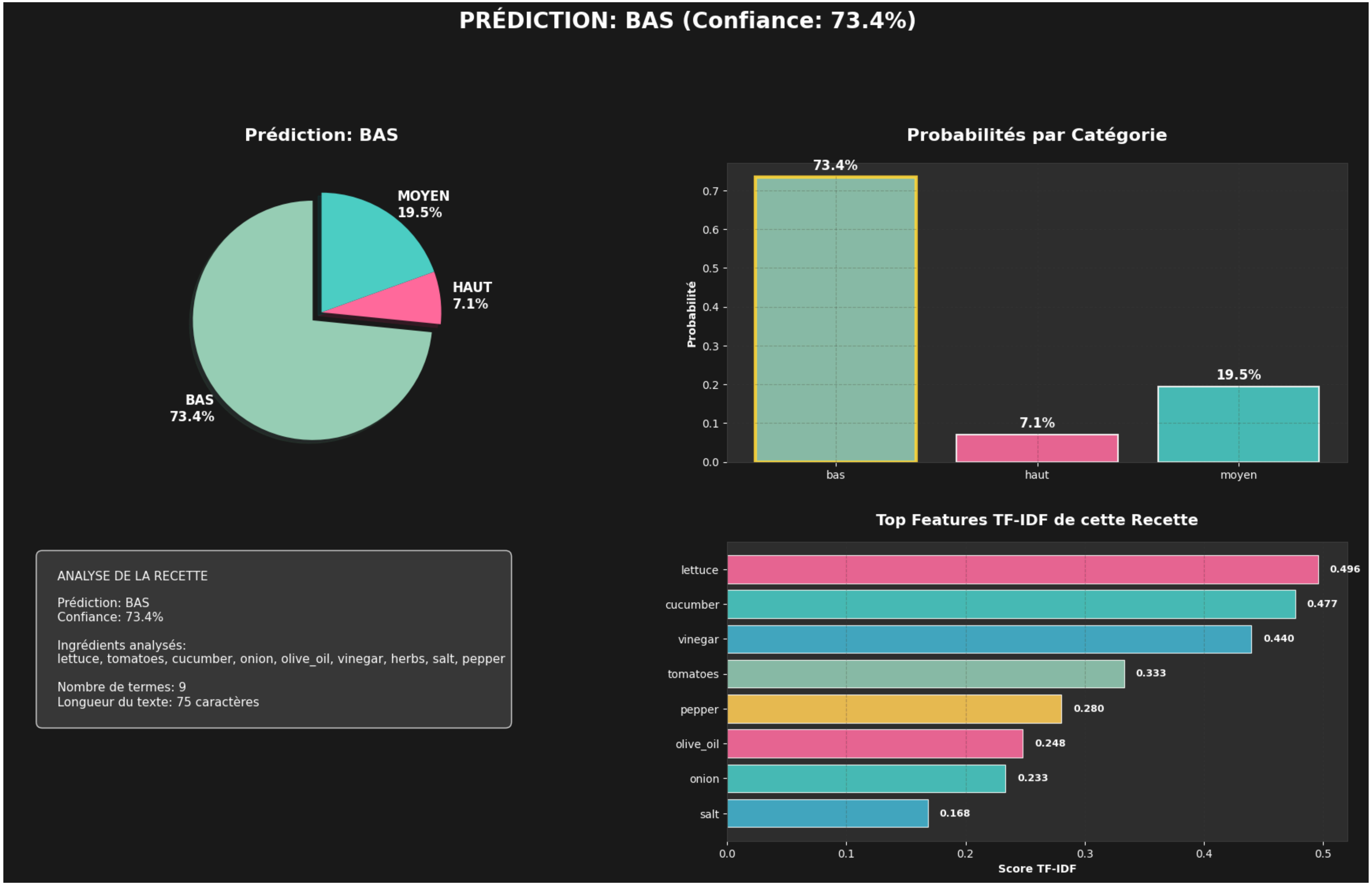


15.4. Test salade légère

In [18]:
```python
# Exemple avec une salade légère
salade_ingredients = "lettuce, tomatoes, cucumber, onion, olive oil, vinegar, herbs, salt, pepper"
```

```python
print(f"\nExemple 2 - Salade légère:")
print(f"Ingrédients: {salade_ingredients}")
pred2, prob2 = visualize_prediction(salade_ingredients)
```

```
Exemple 2 - Salade légère:
Ingrédients: lettuce, tomatoes, cucumber, onion, olive oil, vinegar, herbs, salt, pepper
Original: lettuce, tomatoes, cucumber, onion, olive oil, vinegar, herbs, salt, pepper
Cleaned: lettuce tomatoes cucumber onion olive_oil vinegar herbs salt pepper
Features numériques calculées:
- n_ingredients: 9
- Compteurs: fat=1, sugar=0, protein=0, vegetable=4
- Compteurs: grain=0, drink=0, spice=1
- Ratios: fat=0.111, sugar=0.000, protein=0.000

Dimensions finales:
- TF-IDF: (1, 500)
- Numériques: (1, 15)
- Combinées: (1, 515)
- Modèle attend: 515 features

Résultat:
- Prédiction: bas
- Confiance: 73.4%
- bas: 73.4%
- haut: 7.1%
- moyen: 19.5%
```

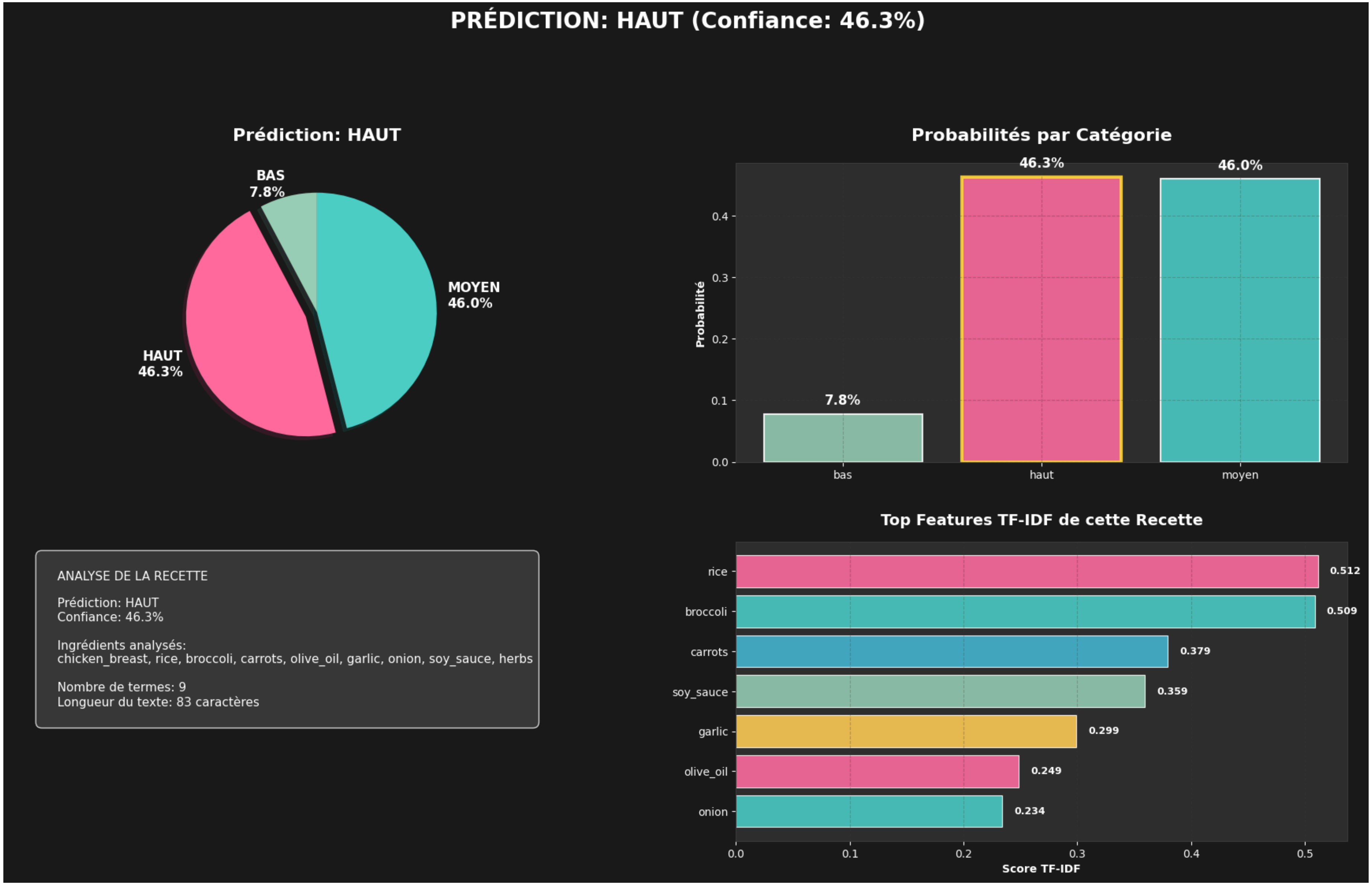


15.5. Test plat équilibré

In [19]:
```
# Exemple avec un plat équilibré
plat_ingredients = "chicken breast, rice, broccoli, carrots, olive oil, garlic, onion, soy sauce, herbs"
```

```
print(f"\nExemple 3 - Plat équilibré:")
print(f"Ingrédients: {plat_ingredients}")
pred3, prob3 = visualize_prediction(plat_ingredients)
```

```
Exemple 3 - Plat équilibré:
Ingrédients: chicken breast, rice, broccoli, carrots, olive oil, garlic, onion, soy sauce, herbs
Original: chicken breast, rice, broccoli, carrots, olive oil, garlic, onion, soy sauce, herbs
Cleaned: chicken_breast rice broccoli carrots olive_oil garlic onion soy_sauce herbs
Features numériques calculées:
- n_ingredients: 9
- Compteurs: fat=1, sugar=0, protein=1, vegetable=4
- Compteurs: grain=1, drink=1, spice=0
- Ratios: fat=0.111, sugar=0.000, protein=0.111

Dimensions finales:
- TF-IDF: (1, 500)
- Numériques: (1, 15)
- Combinées: (1, 515)
- Modèle attend: 515 features

Résultat:
- Prédiction: haut
- Confiance: 46.3%
- bas: 7.8%
- haut: 46.3%
- moyen: 46.0%
```



12. Sauvegarde du modèle

In [ ]:
```
import joblib

# Sauvegarder le modèle et le vectoriseur
joblib.dump(best_xgb, './models/calorie_prediction_model_v1.pkl')
joblib.dump(tfidf, './models/tfidf_vectorizer_v1.pkl')

print("Modèle et vectoriseur sauvegardés!")
print("- ./models/calorie_prediction_model_v1.pkl")
print("- ./models/tfidf_vectorizer_v1.pkl")

# Pour charger plus tard:
# loaded_model = joblib.load('./models/calorie_prediction_model_v1.pkl')
# loaded_tfidf = joblib.load('./models/tfidf_vectorizer_v1.pkl')
```

```
Modèle et vectoriseur sauvegardés!
- ./model/calorie_prediction_model.pkl
- ./model/tfidf_vectorizer.pkl
```

13. Résumé des résultats et conclusions

In [21]:
```
print("=" * 60)
print("RÉSUMÉ DU MODÈLE DE PRÉDICTION CALORIQUE HARMONISÉ")
print("=" * 60)
print(f"Dataset: {df.shape[0]:,} recettes")
print(f"Features: {X_combined.shape[1]:,} features TF-IDF")
print(f"Classes: {sorted(best_xgb.classes_)}")
print(f"\nPerformances:")
print(f"   Accuracy d'entraînement: {train_accuracy:.4f}")
print(f"   Accuracy de test: {test_accuracy:.4f}")
print(f"   Score de validation croisée: {cv_scores.mean():.4f} (+/- {cv_scores.std() * 2:.4f})")
print(f"\nMeilleurs hyperparamètres:")
for param, value in search_xgb_balanced.best_params_.items():
    print(f"   {param}: {value}")
print(f"\nTop 5 features les plus importantes:")
for i, (feature, importance) in enumerate(importance_df.head(5).values):
    print(f"   {i+1}. {feature}: {importance:.4f}")
print("=" * 60)

print("\nCONCLUSIONS PRINCIPALES:")
```

```
print("• Le modèle XGBoost peut prédire efficacement les niveaux caloriques")
print("• Les ingrédients riches (beurre, crème, sucre) sont de bons prédicteurs de calories élevées")
print("• L'analyse SHAP permet de comprendre les contributions de chaque feature")
print("• Le modèle peut être utilisé pour évaluer de nouvelles recettes")
print("• Les bonnes pratiques ML ont été appliquées (nettoyage, validation croisée, optimisation)")
print("• Fonction de prédiction interactive avec analyses détaillées")
print("=" * 60)
```

```
============================================================
RÉSUMÉ DU MODÈLE DE PRÉDICTION CALORIQUE HARMONISÉ
============================================================
Dataset: 228,430 recettes
Features: 515 features TF-IDF
Classes: [0, 1, 2]

Performances:
   Accuracy d'entraînement: 0.6501
   Accuracy de test: 0.5300
   Score de validation croisée: 0.5238 (+/- 0.0027)

Meilleurs hyperparamètres:
   subsample: 0.7
   reg_lambda: 1.5
   reg_alpha: 0.3
   n_estimators: 300
   min_child_weight: 1
   max_depth: 8
   learning_rate: 0.15
   gamma: 0.2
   colsample_bytree: 0.7

Top 5 features les plus importantes:
   1. nb_protein: 0.0106
   2. spaghetti: 0.0073
   3. pork_chops: 0.0073
   4. egg_whites: 0.0067
   5. skim_milk: 0.0066
============================================================

CONCLUSIONS PRINCIPALES:
• Le modèle XGBoost peut prédire efficacement les niveaux caloriques
• Les ingrédients riches (beurre, crème, sucre) sont de bons prédicteurs de calories élevées
• L'analyse SHAP permet de comprendre les contributions de chaque feature
• Le modèle peut être utilisé pour évaluer de nouvelles recettes
• Les bonnes pratiques ML ont été appliquées (nettoyage, validation croisée, optimisation)
• Fonction de prédiction interactive avec analyses détaillées
============================================================
```