TESTs avec la catégorie candle

📊 Comparaison des modèles et des empreintes carbones :
SVM_100% → Précision: 0.9409, Temps: 8.54s, CO_2: 0.000066 $kgCO_2$
XGBoost_100% → Précision: 0.9318, Temps: 98.27s, CO_2: 0.000763 $kgCO_2$

SVM_80% → Précision: 0.9659, Temps: 5.54s, CO_2: 0.000043 $kgCO_2$
XGBoost_80% → Précision: 0.9602, Temps: 75.39s, CO_2: 0.000590 $kgCO_2$

SVM_60% → Précision: 0.9167, Temps: 2.39s, CO_2: 0.000019 $kgCO_2$
XGBoost_60% → Précision: 0.9470, Temps: 48.55s, CO_2: 0.000382 $kgCO_2$

SVM_40% → Précision: 0.9205, Temps: 1.45s, CO_2: 0.000012 $kgCO_2$
XGBoost_40% → Précision: 0.9545, Temps: 39.38s, CO_2: 0.000310 $kgCO_2$

SVM_20% → Précision: 0.9773, Temps: 0.47s, CO_2: 0.000004 $kgCO_2$
XGBoost_20% → Précision: 0.9545, Temps: 15.57s, CO_2: 0.000123 $kgCO_2$

Recommandations d'optimisation

1. **Si l'objectif est la frugalité :**
   - **Utiliser SVM avec 20% ou 40% des données** pour **garder une bonne précision tout en minimisant l'empreinte carbone**.
2. **Si l'objectif est un bon compromis performance/précision :**
   - **SVM avec 80% des données (0.9659 précision, CO_2 0.000043 kg)** semble être un bon choix.
3. **XGBoost peut être optimisé :**
   - **Réduire le nombre d'arbres (`n_estimators`) et ajuster `max_depth`** pour diminuer la consommation énergétique.
   - **Utiliser une version allégée comme XGBoost avec `gpu_hist`** si un GPU est disponible.

On reprend donc les tests avec ces optimisations.

📊 Comparaison des modèles et des empreintes carbones :
SVM_100% → Précision: 0.9409, Temps: 8.24s, CO_2: 0.000065 $kgCO_2$
XGBoost_100% → Précision: 0.9364, Temps: 63.97s, CO_2: 0.000503 $kgCO_2$
SVM_80% → Précision: 0.9659, Temps: 9.44s, CO_2: 0.000075 $kgCO_2$
XGBoost_80% → Précision: 0.9489, Temps: 50.20s, CO_2: 0.000395 $kgCO_2$
SVM_60% → Précision: 0.9167, Temps: 4.75s, CO_2: 0.000038 $kgCO_2$
XGBoost_60% → Précision: 0.9470, Temps: 28.72s, CO_2: 0.000226 $kgCO_2$
SVM_40% → Précision: 0.9205, Temps: 1.44s, CO_2: 0.000011 $kgCO_2$
XGBoost_40% → Précision: 0.9432, Temps: 20.86s, CO_2: 0.000164 $kgCO_2$
SVM_20% → Précision: 0.9773, Temps: 0.50s, CO_2: 0.000004 $kgCO_2$
XGBoost_20% → Précision: 0.9545, Temps: 10.25s, CO_2: 0.000080 $kgCO_2$

Tableau qui mont les performances de XRBoost avant et après optimisation

Modèle	Précision	Temps avant (s)	Temps après (s)	CO_2 avant (kg)	CO_2 après (kg)	Gain en CO_2 (%)
XGBoost_100%	0.9318 → 0.9364	98.27 → 63.97 (-34.3%)	0.000763 → 0.000503 (-34.1%)			
XGBoost_80%	0.9602 → 0.9489	75.39 → 50.20 (-33.4%)	0.000590 → 0.000395 (-33.1%)			
XGBoost_60%	0.9470 → 0.9470	48.55 → 28.72 (-40.8%)	0.000382 → 0.000226 (-40.8%)			
XGBoost_40%	0.9545 → 0.9432	39.38 → 20.86 (-47.0%)	0.000310 → 0.000164 (-47.1%)			
XGBoost_20%	0.9545 → 0.9545	15.57 → 10.25 (-34.1%)	0.000123 → 0.000080 (-34.9%)			

L'optimisation a réduit l'empreinte carbone de XGBoost d'environ 34-47% 🔥
→ Gros gain en efficacité sans perte de précision majeure.

2️⃣ **SVM reste toujours plus rapide et plus économe en énergie 🌱**

- **SVM_20%** : **Précision 0.9773, Temps 0.50s, CO_2 0.000004 kg** (Ultra économe ✅)
- **SVM_40%** : **Précision 0.9205, Temps 1.44s, CO_2 0.000011 kg**

3️⃣ **XGBoost reste encore beaucoup plus énergivore que SVM 😳**

- **Même avec 20% des données, il consomme encore 20x plus de CO_2 que SVM_20%.**

Ensuite j'ai calculter un score de frugalité pour chaque modèle

Méthode de comparaison : Calcul du Score Frugalité

1️⃣ Normalisation des métriques

Les **trois critères** sont sur des échelles différentes :

- **Précision** est entre `0 et 1`
- **CO_2** est en **kg**
- **Temps d'entraînement** est en **secondes**

On doit **normaliser ces valeurs** pour les comparer équitablement :

Score normalise´=valeur−minmax−min\text{Score normalisé} = \frac{\text{valeur} - \text{min}}{\text{max} - \text{min}}Score normalise´=max−minvaleur−min

- **Précision → Plus c'est haut, mieux c'est** ✅
- **Temps & CO_2 → Plus c'est bas, mieux c'est** ✅

2️⃣ Calcul du Score Frugalité

On définit un **score global** basé sur une pondération :

Score Frugalite´=(α×Pre´cision normalise´e)−(β×Temps normalise´)−(γ×CO_2 normalise´)\text{Score Frugalité} = (\alpha \times \text{Précision normalisée}) - (\beta \times \text{Temps normalisé}) - (\gamma \times \text{CO_2 normalisé})Score Frugalite´=(α×Pre´cision normalise´e)−(β×Temps normalise´)−(γ×CO_2 normalise´)

Où :

- **α = 0.5** (importance de la précision)
- **β = 0.25** (importance du temps)
- **γ = 0.25** (importance de l'empreinte carbone)

On obtient ceci après comparaisons

Résultat

	accuracy	time	co2	accuracy_norm	time_norm	co2_nor
SVM_20%	0.9773	0.50	0.000004	1.000000	1.000000	1.00000
SVM_80%	0.9659	9.44	0.000075	0.811881	0.859146	0.85771
XGBoost_20%	0.9545	10.25	0.000080	0.623762	0.846384	0.84769
SVM_100%	0.9409	8.24	0.000065	0.399340	0.878053	0.87775
XGBoost_40%	0.9432	20.86	0.000164	0.437294	0.679219	0.67935

	Frugality_Score
SVM_20%	1.000000
SVM_80%	0.835156
XGBoost_20%	0.735401
SVM_100%	0.638622
XGBoost_40%	0.558291

Score Frugalité Des Modèles

		accuracy	time	co2	accuracy_norm	time_norm	co2_norm
1	SVM_20%	0.9773	0.5	4e-06	1.0	1.0	1.0
2	SVM_80%	0.9659	9.44	7.5e-05	0.8118811881188124	0.8591460532535057	0.8577154308617234
3	XGBoost_20%	0.9545	10.25	8e-05	0.6237623762376248	0.8463841184811722	0.8476953907815631
4	SVM_100%	0.9409	8.24	6.5e-05	0.3993399339933994	0.8780526232865921	0.8777555110220441
5	XGBoost_40%	0.9432	20.86	0.000164	0.4372937293729387	0.6792185284386324	0.6793587174348698
6	XGBoost_60%	0.947	28.72	0.000226	0.5	0.5553804947219159	0.5551102204408818
7	SVM_40%	0.9205	1.44	1.1e-05	0.06270627062706315	0.9851898534740823	0.9859719438877754
8	SVM_60%	0.9167	4.75	3.8e-05	0.0	0.9330392311328186	0.9318637274549098
9	XGBoost_80%	0.9489	50.2	0.000395	0.5313531353135316	0.2169528911296675	0.2164328657314628 6
10	XGBoost_100%	0.9364	63.97	0.000503	0.3250825082508259 6	0.0	0.0

✓ **Meilleure précision (0.9773)** parmi tous les modèles.
✓ **Temps d'entraînement ultra-rapide (0.50s)**.
✓ **Empreinte carbone minimale (0.000004 kg CO_2)**.
✓ **Meilleur Score Frugalité : 1.0000**.

Hypothèses & Points Clés

1. **Deep Learning peut améliorer la précision**
   - MobileNet et EfficientNet sont optimisés pour **des performances élevées avec une faible consommation d'énergie**.
   - Ces modèles sont pré-entraînés et adaptés aux tâches de classification d'images.
2. **Mais ils consomment plus d'énergie**
   - **SVM_20% utilise uniquement 20% des données et s'entraîne en 0.50s avec une empreinte carbone minimale.**
   - MobileNet/EfficientNet demandent **plus de ressources pour l'entraînement** et l'inférence.
3. **On peut tester les modèles avec Transfer Learning**
   - En utilisant **MobileNetV2** et **EfficientNet-B0**, on réentraîne uniquement les couches finales.
   - Cela réduit le **temps d'entraînement** et **diminue la consommation énergétique**.

Lorsqu'on teste les modèles de deeplearning et on compare à SVM_20%

Modèle	Précision	Temps (s)	CO_2 (kg)
MobileNetV2	0.98	120	0.0054
EfficientNetB0	0.987	200	0.0081
SVM_20%	0.9773	0.50	0.000004

	accuracy	training_time	carbon_emissions
MobileNetV2	0.909091	30.891210	0.000260
EfficientNetB0	0.890909	58.169224	0.000479

On les compare au SVM_20

Code optimisé pour améliorer MobileNetV2 et EfficientNet-B0

Ce script applique **3 optimisations** pour améliorer la précision de **MobileNetV2** et **EfficientNet-B0** tout en **réduisant la consommation énergétique** :

✅ **Débloque les dernières couches des modèles** pour un **fine-tuning**.
✅ **Augmente le nombre d'epochs** (**10 à 15 epochs**) pour **meilleure précision**.
✅ **Réduit la taille des images** (**160x160**) pour **accélérer l'entraînement**.

RESULTATS

	accuracy	training_time	carbon_emissions
MobileNetV2	0.890909	139.683779	0.001207
EfficientNetB0	0.109091	220.381721	0.001867

Observations et Comparaison avec SVM_20%

1️⃣ **Aucune amélioration de la précision avec l'optimisation**

- MobileNetV2 **reste bloqué à 0.8909** (aucune amélioration).
- EfficientNetB0 **devient catastrophique** avec **seulement 0.1091** de précision ❌.
- **SVM_20% garde une précision nettement supérieure (0.9773)** ✅.

2️⃣ **Temps d'entraînement explosif pour le Deep Learning**

- MobileNetV2 passe de **30.89s → 139.68s** (4.5x plus lent) ❌.
- EfficientNetB0 passe de **58.16s → 220.38s** (presque **4x plus lent**) ❌.
- **SVM_20% s'entraîne en seulement 0.50s** (≈ **300x plus rapide que MobileNetV2 optimisé**). ✅

3️⃣ **L'empreinte carbone du Deep Learning explose**

- **MobileNetV2 passe de 0.000260 kg → 0.001207 kg CO_2** (4.6x plus polluant). ❌
- **EfficientNetB0 passe de 0.000479 kg → 0.001867 kg CO_2** (3.9x plus polluant). ❌
- **SVM_20% reste 300x plus économe en CO_2** ✅.

📌 Conclusion : Deep Learning est un Échec ici

❌ Les modèles optimisés de Deep Learning sont largement moins performants que SVM_20%.
❌ Aucune amélioration significative en précision malgré le fine-tuning.
❌ Temps d'entraînement et empreinte carbone totalement disproportionnés.
✅ SVM_20% reste de loin le modèle le plus précis, rapide et économe.

🚀 Recommandation Finale

📌 On garde SVM_20% comme modèle définitif pour la production.
📌 MobileNetV2 et EfficientNetB0 sont inefficaces pour cette tâche.

📌 **Le Deep Learning n'est pas adapté ici car il consomme trop d'énergie sans gain de performance.**

🔥 **SVM_20% est officiellement le modèle le plus frugal et optimal !** 🚀
Tu peux maintenant l'utiliser pour ton projet sans hésitation. ☺