Challenge Python

▼ Description du challenge

le challenge consiste à créer un modèle de machine learning pour prédire la qualité d'un téléphone portable. Les données sont stockées dans un DataFrame et contiennent des informations sur diverses caractéristiques des téléphones portables, telles que la puissance de la batterie, la résolution de l'écran, la quantité de mémoire RAM, etc.

L'objectif est de construire un modèle capable de prédire la qualité d'un téléphone portable en fonction de ces caractéristiques, qui sont les variables explicatives. La qualité est représentée par une valeur entière allant de 0 (mauvaise qualité) à 3 (excellente qualité).

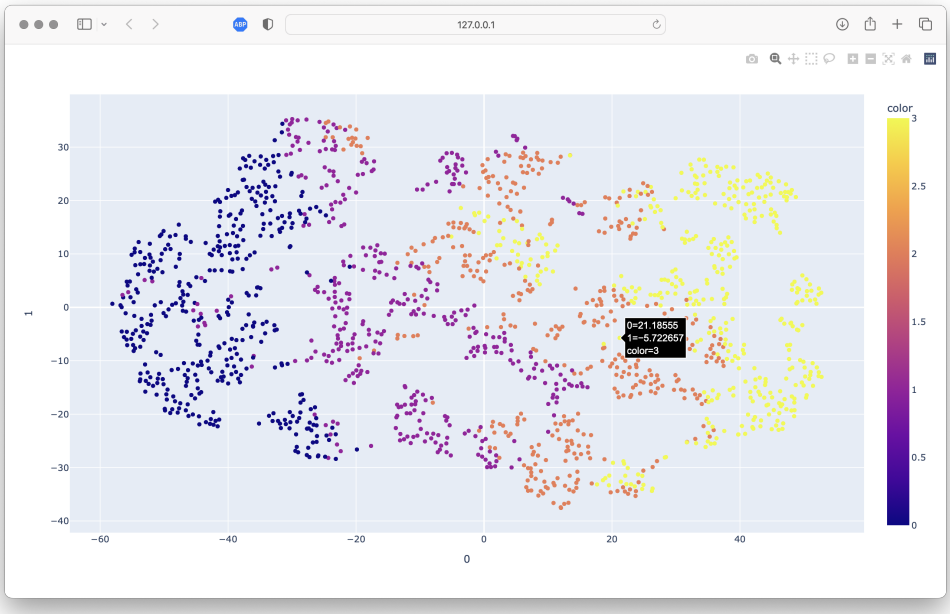
Pour atteindre cet objectif, nous avons utilisé plusieurs techniques de machine learning, telles que la régression logistique, l'arbre de décision et le réseau de neurones artificiels. Nous avons également effectué une recherche d'hyper-paramètres pour chaque technique pour trouver les meilleurs paramètres pour notre modèle.

Enfin, nous avons évalué la performance de chaque modèle en utilisant la précision comme métrique d'évaluation.

▼ Répartition des données

code pour récupérer et afficher la répartition de chaque points de notre df d'entrainement

```
df = pd.read_csv("mobile_train.csv")
########## On regarde la répartition des classes ##########

features = df.columns[:-1]
tsne = TSNE(n_components=2, random_state=42)
X_embedded = tsne.fit_transform(df[features].values)
fig = px.scatter(X_embedded, x=0, y=1, color=df['price_range'])
fig.show()
```



On peut voir que la répartition n'est pas trop mal et que la méthode de k plus proches voisins est envisageable

▼ Méthode k plus proche voisins

▼ Implémentation de l’algorithme k plus proche voisins

```
##### modèle KNN vu en cours #####

########## On définit la fonction de distance ##########

def distance_euclidienne(v1, v2):
    return np.sqrt(np.sum((v1 - v2) ** 2))


########## On définit la fonction de prédiction ##########

def neighbors(X_train, y_label, x_test, k):
    list_distances = []
    for i in range(np.shape(X_train)[0]):
        list_distances.append(distance_euclidienne(X_train.iloc[i], x_test))
    dataf = pd.DataFrame()
    dataf["label"] = y_label
    dataf["distance"] = list_distances
    dataf = dataf.sort_values(by="distance")
    return dataf.iloc[:k, :]


def predict(neighbors):
    return neighbors["label"].value_counts().idxmax()


def evaluation(X_train, Y_train, X_valid, Y_valid, k, verbose=True):
    TP = 0  # vrai
    FP = 0  # faux
    total = 0
    for i in range(X_valid.shape[0]):
        nearest_neighbors = neighbors(X_train, Y_train, X_valid.iloc[i], k)
        if predict(nearest_neighbors) == Y_valid.iloc[i]:
            TP += 1
        else:
            FP += 1
        total += 1
    accuracy = TP / total
    if verbose:
        print("Accuracy:" + str(accuracy))
    return accuracy
```

les premiers essais avec 5 voisins donnent des résultats plutôt correcte autour de 70%

on va maintenant essayer de chercher de manière plus optimiser le meilleur nombre voisins à prendre

▼ Recherche du meilleur nombre de voisins

Voici une fonction pour voir une moyenne de la précision de l’algorithme des kppv de la bibliothèque

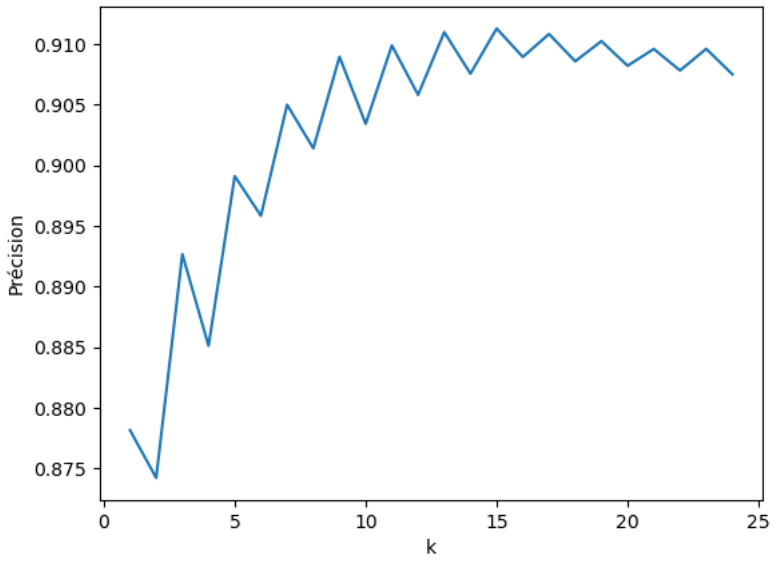
`from sklearn.neighbors import KNeighborsClassifier` (entre 1 et x voisins)

sur x iterations

```
##### Définir le meilleur nombre de voisin avec la bibliothèque KNeighborsClassifier de sklearn.neighbors #####

def meilleurNbrVoisin(df, n_iterations,nbrVoisinMax):

    # Stocker les précisions de chaque itération dans une liste
    all_accuracies = []
    for i in range(n_iterations):
        digits_train = df.sample(frac=0.8)
        digits_valid = df.drop(digits_train.index)

        X_train = digits_train[features]
        Y_train = digits_train['price_range']
        X_valid = digits_valid[features]
        Y_valid = digits_valid['price_range']
        list_accuracy = []
        for k in range(1, nbrVoisinMax):
            knn = KNeighborsClassifier(n_neighbors=k)

            # Entraîner le modèle sur les données d'entraînement
            knn.fit(X_train, Y_train)

            # Prédire les étiquettes pour les données de validation
            Y_pred = knn.predict(X_valid)
            accuracy = sum(Y_pred == Y_valid) / len(Y_valid)
            list_accuracy.append(accuracy)
        all_accuracies.append(list_accuracy)
```

```
    # Calculer la moyenne et l'écart-type de toutes les précisions pour chaque valeur de k
    mean_accuracies = np.mean(all_accuracies, axis=0)
    std_accuracies = np.std(all_accuracies, axis=0)

    # Tracer la courbe de précision moyenne avec des barres d'erreur
    plt.plot(range(1, nbrVoisinMax), mean_accuracies)
    plt.xlabel('k')
    plt.ylabel('Précision')
    plt.show()
```

voici le graphique obtenues entre 1 et 25 voisins avec 2000 itérations



On peut en déduire que le meilleur nombre de voisins à prendre est de 15

▼ Test sur les vrai données

voici la fonction qui me permet de créer le csv remplis des données de prédictions pour le df de test

```
########## test sur le vrai modèle ##########
def testModeleKNN(df,nbrVoisin):
    df2 = pd.read_csv("mobile_test_data.csv")

    knn1 = KNeighborsClassifier(n_neighbors=nbrVoisin)
    X = df2[features]
    # Entraîner le modèle sur les données d'entraînement
    knn1.fit(df[features], df['price_range'])

    # Prédire les étiquettes pour les données de validation
    Y_pred = knn1.predict(X)
    print(Y_pred)
    np.savetxt("test_mobile_predictions.csv", Y_pred, delimiter="\n")
```

avec la méthode de K plus proche voisins on se retrouve à obtenir un score de 94,75%

On va maintenant chercher si des labels inutiles peuvent être supprimer pour affiner la recherche

▼ Recherche des label inutile à la recherche

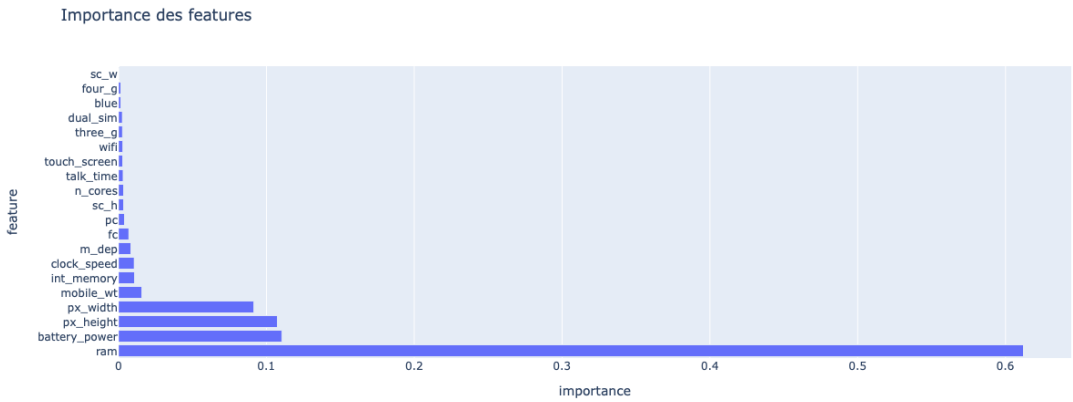
voici la fonction que j'ai faite pour tester l'importance des labels grace à la librairie

'DecisionTreeClassifier' de 'sklearn.tree' et qui affiche le pourcentage d'importance pour chaque labels et qui ajoute les 4 meilleurs dans 'liste_meilleur_label'

on verra juste après pourquoi les 4 meilleurs

```
from sklearn.tree import DecisionTreeClassifier
def importanceLabel(liste_meilleur_label):
    # Diviser les données en ensemble d'entraînement et ensemble de validation
    X_train, X_valid, Y_train, Y_valid = train_test_split(df[features], df['price_range'], test_size=0.2, random_state=42)

    # Créer un arbre de décision
    tree = DecisionTreeClassifier(random_state=42)

    # Entraîner l'arbre de décision sur les données d'entraînement
    tree.fit(X_train, Y_train)

    # Obtenir les importances des features
    importances = tree.feature_importances_

    # Trier les labels par ordre d'importance décroissante
    indices = np.argsort(importances)[::-1]

    # Créer un DataFrame contenant les labels et leurs importances
    df_importances = pd.DataFrame({'feature': X_train.columns[indices], 'importance': importances[indices]})
    for i in range(0, 4):
        liste_meilleur_label.append(df_importances['feature'][i])
    # Créer un graphique à barres pour visualiser les importances des features
    fig = px.bar(df_importances, x='importance', y='feature', orientation='h', height=500, title='Importance des features')
    fig.show()
```

voici le graphique obtenue et on voit que 4 labels ressortent du lot et c'est donc pour ça que j'ai choisis pour la suite de garder uniquement ces 4 labels



pour la suite on va maintenant s'intéresser à la méthode du réseau de neurone Perceptron

▼ Méthode du réseau de neurone Perceptron

Voici la fonction d'évaluation que j'utiliserai pour tester mes différents essaie de paramètre de réseau de neurone

```
def evaluation2(X_valid, Y_valid):
    clf = MLPClassifier(solver='adam', alpha= 1e-05, hidden_layer_sizes= (10), learning_rate= 'adaptive', max_iter=2000)
    clf.fit(X_train, Y_train)
    accuracy = clf.score(X_valid, Y_valid)
    print("Accuracy:", accuracy)
```

Après un premier essai avec les dernières valeurs utiliser en cours j'obtiens une accuracy autour 25% ce qui est très peu je décide donc de normaliser les données et re tester

▼ Normalisation des données

Pour normaliser mes données j'utiliserai une autre bibliothèque de 'sklearn' 'MinMaxScaler'

```
##### Normalisation #####
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

```
X_trainN = scaler.fit_transform(X_train)
X_validN = scaler.fit_transform(X_valid)
XN = scaler.fit_transform(X[liste_meilleur_label])
```

après essai sur uniquement les labels intéressant et en normalisant je me retrouve avec une accuracy autour de 90%

Je décide donc maintenant de me pencher sur la recherche des meilleur paramètre à prendre dans mon réseau de neurones

▼ Recherche meilleur hyper-paramètre

voici la fonction qui cherche le meilleur couple de paramètre possible parmi une liste d'hidden_layer_sizes, du coefficient alpha et des différents learning_rate que propose le solver adam

```
def rechercheMeilleurHyperparametre():
    # Définir les valeurs des hyperparamètres à tester
    param_grid = {
        'hidden_layer_sizes': [(10,10,10,10,10,10,10,10,10,10),(20,20,20,20,20,20,20,20),
                               (50,50,50,50,50),(150),(10,10,10),(30,30,30),(10,10,10,10,10),(10,10),(30,30)],
        'alpha': [0.01,0.001,0.0001, 0.00001, 0.000001],
        'learning_rate': ['constant', 'invscaling', 'adaptive', 'learning_rate_init']

    }

    # Créer un objet GridSearchCV pour tester toutes les combinaisons des hyperparamètres
    grid_search = GridSearchCV(
        MLPClassifier(solver='adam', learning_rate='adaptive', max_iter=2000),
        param_grid,
        cv=7, # nombre de folds de la validation croisée
        n_jobs=-1, # utiliser tous les coeurs du processeur
        verbose=2, # afficher les détails de chaque itération
    )

    # Exécuter la recherche des meilleurs hyperparamètres sur les données d'entraînement
    grid_search.fit(XN, Y)

    # Afficher les meilleurs hyperparamètres et la précision correspondante
    print(f"Meilleurs hyperparamètres : {grid_search.best_params_}")
    print(f"Précision correspondante : {grid_search.best_score_}")
```

voici ce que nous donne cette fonction

```
Meilleurs hyperparamètres : {'alpha': 1e-05, 'hidden_layer_sizes': (10, 10, 10), 'learning_rate': 'adaptive'}
Précision correspondante : 0.9487665670727036
```

et après essaie de ces meilleurs hyper-paramètres dans la fonction evaluation2 on se retrouve avec une accuracy de 92%

▼ Test sur les vrais données

voici de le code qui me permettra de stocker mes prédictions avec le modèle de réseau de neurones dans un csv

```
########## test sur le vrai modèle ##########
def testModeleMLP():
    df2 = pd.read_csv("mobile_test_data.csv")

    clf = MLPClassifier(solver='adam', alpha= 1e-05, hidden_layer_sizes= (10, 10, 10), learning_rate= 'adaptive', max_iter=2000
    X2 = df2[liste_meilleur_label]
    X2N = scaler.fit_transform(X2[liste_meilleur_label])
    # Entraîner le modèle sur les données d'entraînement
    clf.fit(XN, Y)

    # Prédire les étiquettes pour les données de validation
    Y_pred = clf.predict(X2N)
    print(Y_pred)
    np.savetxt("mobile_test_predictions.csv", Y_pred, delimiter="\n")
```

après essai on se retrouve avec une accuracy de 95.25%