Devoir 2 : Intervalles de confiance et régression

Le staff de MATH0487-2

Décembre 2023

Nom(s), Prénom(s) et matricule(s) :

DUCHEMIN Catherine s202046

BOUSTANI Mehdi s221594

Instructions générales

Objectifs Les objectifs de ce devoir sont les suivants :

- estimer un intervalle de valeurs pour un paramètre à un niveau de confiance donné
  - en faisant une approximation normale,
  - en utilisant la méthode du *bootstrapping*,
- estimer la valeur des paramètres d'une régression linéaire à partir de données réelles, analyser la qualité du modèle de régression et identifier ses potentiels défauts ou les hypothèses qui ne sont pas respectées, et réintroduire la notion de leverage,
- appliquer un modèle de régression logistique sur des données réelles et analyser la qualité du modèle de régression logistique.

Délivrables Ce devoir doit être réalisé par groupe de 2 étudiants maximum. Chaque groupe doit rendre ce notebook complété, et **une version .pdf de ce notebook** qui sera utilisée pour la correction.

La date limite de soumission est fixée au **20 décembre 2023 à 20h00**. Jusqu'à cette date, vous avez la possibilité de (re)soumettre votre rapport ou votre code autant de fois que vous le souhaitez. Au-delà de cette date, il ne sera plus possible de soumettre le devoir. N'attendez pas la dernière minute pour soumettre une première version de votre travail !

La soumission doit se faire sur la plateforme Gradescope directement.

- Chaque étudiant doit s'inscrire sur Gradescope en utilisant son adresse `@student.uliege.be`. Si vous ne voyez pas le cours MATH0487 dans votre tableau de bord, contactez-nous sur Ed au plus vite (n'attendez pas la veille de la date de soumission pour vérifier que vous avez accès au cours sur Gradescope ;-).
- Chaque groupe doit soumettre un seul fichier `.ipynb` et un seul fichier `.pdf` sur Gradescope. Toutes les cellules doivent être exécutables et leurs sorties ne doivent pas être effacées avant la soumission. Assurez-vous que tous les membres du groupe sont correctement ajoutés à la soumission !

- N'oubliez pas d'assigner les pages de votre pdf aux questions sur Gradescope !

Si vous n'êtes pas familiers avec Gradescope, vous trouverez des explications sur chaque étape de la soumission ci-dessous :

- Soumission de pdf,
- Soumission de code,
- Ajout de membres de groupe.

Pour convertir votre notebook en pdf, nous conseillons l'utilisation de nbconverter. Cet outil est également utilisé par JupyterLab pour l'exportation de notebooks.

Remarques importantes sur l'utilisation de ce notebook :

- Ne modifiez et ne supprimez pas de cellules (Markdown) contenant des consignes/questions.
- Les cellules demandant une réponse sous forme de texte ou sous forme de code sont colorées en vert,
  comme ceci.
- Remplissez uniquement les cellules prévues à cet effet :
  - `python """VOTRE CODE"""` indique une portion réservée à votre code, et
  - VOTRE TEXTE ICI
    indique une portion réservée à une réponse écrite. N'en créez pas de nouvelles.
- Respectez le type de cellule prévu pour une question donnée: certaines questions demandent d'implémenter du code (cellules "Code", en Python) et de présenter des résultats (valeurs numériques, tables, graphes, ...), et d'autres vous demandent de fournir une réponse utilisant du texte (cellules "Markdown", incluant certaines commandes LaTeX et acceptant la syntaxe HTML).
- Lorsque vous présentez un graphique, n'oubliez pas d'indiquer un titre et / ou des noms pour les axes. Lorsque cela est nécessaire, affichez également la légende.
- Un exemple de cellule Markdown comprenant des commandes LaTex est donné ci-dessous :

début de l'exemple

Ceci est un *exemple* de cellule de texte **Markdown**. Double-cliquez dessus pour voir le texte brut.

Vous pouvez utilisez certaines commandes LaTeX comme $\sin(x)$ ou α. Il est possible d'écrire des équations comme

$$\beta\dot{y} = 3x$$

ou encore

$$\begin{aligned} \beta\dot{y} &= 3x \\ \gamma\dot{x} &= 4y. \end{aligned} \tag{1}$$

Des listes sont également disponibles :

- élément 1
- élément 2
- ...

ou

- élément 1
- élément 2

- …

N'hésitez pas à vous renseigner sur la syntaxe Markdown si vous avez besoin d'autres éléments (*e.g.* construire des tables).

fin de l'exemple

- Enfin, veillez à toujours présenter vos résultats en sortie de cellule quand cela est nécessaire (figures, valeurs numériques, etc.). **Une cellule non exécutée ou dont les valeurs calculées et demandées ne sont pas affichées sera considérée comme non implémentée.**

Si vous rencontrez des problèmes ou avez des questions concernant ces remarques, merci de contacter l'équipe pédagogique *via* le forum de Ed Discussion.

Questions Toutes vos questions sur le devoir (y compris sur l'utilisation de Python ou de Jupyter) doivent être postées dans le forum de Ed Discussion du cours sous la catégorie *Assignments/Homework* (une question par fil de discussion).

Politique de collaboration Vous pouvez discuter du devoir avec d'autres groupes, mais *vous devez écrire vous-même vos propres solutions, et écrire et exécuter vous-même votre propre code.* Copier la solution de quelqu'un d'autre, ou simplement apporter des modifications triviales pour ne pas copier textuellement, n'est pas acceptable.

Présentation du problème À l'occasion de ce second projet, l'équipe de MATH0487 a décidé de lever les yeux vers le ciel, et plus précisément en direction des étoiles. Ce projet porte sur l'estimation par intervalle et la régression (linéaire et logistique), en utilisant des données obtenues grâce à la technique de la parallaxe.

Les astronomes utilisent la méthode de la parallaxe pour calculer la distance des corps célestes situés à une distance modérée de la Terre. Cette méthode consiste à observer les angles de vue depuis deux points distincts, séparés par une distance connue, afin de déterminer la distance d'un astre. Ainsi, plus l'angle de parallaxe est petit, plus l'astre est éloigné.

Des explications plus détaillées et diverses illustrations sont disponibles sur Wikipedia.

L'ensemble des données peut à nouveau être chargé depuis le fichier `data_math0487.csv` de l'archive fournie.

[2]:
```
# ces librairies devraient suffire à réaliser ce second devoir, vous pouvez␣
↪évidemment en utiliser d'autres (à importer dans cette cellule).
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.metrics import mean_squared_error
```

[3]:
```
# Vous pouvez soit importer toutes vos fonctions, les réécrire dans cette␣
↪cellule ou coder sans les fonctions dans les cellules ci-dessous.
def get_normal_CI(data: np.array, confidence:float)->tuple[float, float]:
    """
    Computes an approximate normal confidence interval for the mean of the␣
↪given data.

    Args:
    -----
        - `data` (np.array): A 1-dimensional array containing the data for␣
↪which the confidence interval is to be calculated.
        - `confidence` (float): Confidence level, representing 1-alpha (e.g., 0.
↪95 for a 95% confidence interval).

    Returns:
    --------
        tuple[float, float]: A tuple containing the lower and upper bounds of␣
↪the confidence interval.
    """
    mean = np.mean(data)
    std = np.std(data,  ddof=1) #ddof = 1 pour avoir (n-1) au dénominateur
    N = data.size

    intervalle = stats.norm.interval(confidence, loc=mean, scale=std/(np.
↪sqrt(N)))

    return (intervalle[0], intervalle[1])


def get_bootstrap(data: np.array, confidence:float, n_resamples:int, fun = np.
↪mean)->tuple[float, float, np.array]:
    """
    Computes a confidence interval estimation using the bootstrap method␣
↪(percentile method).

    Args:
    -----
        - `data` (np.array): A 1-dimensional array containing the data for␣
↪which the confidence interval is to be calculated.
        - `confidence` (float): Confidence level, representing 1-alpha (e.g., 0.
↪95 for a 95% confidence interval).
        - `n_resamples` (int): The number of bootstrap resamples to generate.
        - `fun` (callable, optional): A function applied to each resample to␣
↪calculate the statistic of interest. Defaults to `np.mean`.

    Returns:
```

```
    --------
        tuple[float, float, np.array]: A tuple containing the lower and upper␣
 ↪bounds of the confidence interval.
                                       It also contains the distribution as a␣
 ↪numpy array.
    """

    boot_samples = stats.bootstrap((data,), fun, n_resamples=n_resamples,
                                   vectorized=False, axis=0,
                                   confidence_level=confidence,
                                   method='percentile')

    return (boot_samples.confidence_interval.low,
            boot_samples.confidence_interval.high,
            boot_samples.bootstrap_distribution)


def get_linear_model(data: np.array, y: np.array)->tuple:
    """
    Fits a linear regression model using the provided data and observed values,
    and makes predictions on the training data.

    Args:
    -----
        - `data` (np.array): A 2-dimensional array containing the predictive␣
 ↪variables (features).
        - `y` (np.array): A 1-dimensional array containing the observed values␣
 ↪(target variable).

    Returns:
    --------
        tuple:
            - sklearn.linear_model.LinearRegression: The fitted linear␣
 ↪regression model.
            - np.array: Predictions made by the model on the training data.

    """

    return None


def get_residue(y: np.array, y_pred: np.array)->np.array:
    """
    Computes the residuals (differences) between observed values and model␣
 ↪predictions.
```

```
    Args:
    -----
        - `y` (np.array): A 1-dimensional array of observed values (ground␣
 ↪truth).
        - `y_pred` (np.array): A 1-dimensional array of predicted values from a␣
 ↪model.

    Returns:
    --------
        np.array: A 1-dimensional array containing the residuals.
    """

    return None


def get_logistic_model(data: np.array, y: np.array)->tuple:
    """
    Fits a logistic regression model using the provided data and observed␣
 ↪values,
    and makes predictions on the training data.

    Args:
    -----
        - `data` (np.array): A 2-dimensional array containing the predictive␣
 ↪variables (features).
        - `y` (np.array): A 1-dimensional array containing the observed values␣
 ↪(target variable).

    Returns:
    --------
        tuple:
            - sklearn.linear_model.LogisticRegression: The fitted logistic␣
 ↪regression model.
            - np.array: Predictions made by the model on the training data.
    """

    return None


def get_leverage(X: np.array)->np.array:
    """
    Computes the leverage for each crystallisation of the predictive variables.

    Args:
    -----
```

```
        - `X` (np.array): A 2-dimensional array representing the matrix of␣
↪crystallisations
                    of the predictive variables (features).

    Returns:
    --------
        np.array: A 1-dimensional array containing the leverage values for each␣
↪crystallisation.
    """

    return None


def get_specific_residue_leverage(X: np.array, y:np.array, x_pos:np.array,␣
↪y_pos:np.array)->tuple[np.array, np.array]:
    """
    Computes the residuals and leverage for a group of specific␣
↪cristallisations to be added to
    the initial dataset.

    Args:
    -----
        X (np.array): A 2-dimensional array representing the initial matrix of
                      crystallisations of the predictive variables (features).
        y (np.array): A 1-dimensional array of the initial observed variables␣
↪(target values).
        x_pos (np.array): A 1-dimensional array of predictive variable values␣
↪to be added to `X` (only the features, no bias in the argument).
        y_pos (np.array): A 1-dimensional array of observed variable values to␣
↪be added to `y`.

    Returns:
    --------
        tuple[np.array, np.array]:
            - np.array: Residuals for each position specified by `x_pos` and␣
↪`y_pos`.
            - np.array: Leverage values for each position specified by `x_pos`␣
↪and `y_pos`.
    """

    return None
```

Importation des données

[4]:
```
data = pd.read_csv('data_math0487.csv')
data.head(5)
```

[4]:

	Vmag	Plx	B-V	Amag	TargetClass
0	5.99	13.592265	1.318	16.678352	0
1	8.70	2.196457	-0.045	15.518060	0
2	5.77	5.802407	0.855	14.471813	0
3	6.72	5.989588	-0.015	15.324928	1
4	8.76	14.067170	0.584	19.401997	1

Le dataset fourni est structuré de la manière suivante : - `Vmag` : la magnitude apparente de l'étoile en question, - `Plx` : angle de la parallaxe, - `B-V` : Indice de couleur de l'étoile, - `Amag` : la magnitude absolue de l'étoile, - `Target Class` : booléen caractérisant une étoile en tant que naine ou géante.

Dans la pratique, les observations fournies par les appareils de mesure ne sont jamais parfaites : il y a toujours du bruit. Dans ce contexte, la source principale du bruit provient de la parallaxe. (*Indice* : cela pourrait vous aider à interpréter vos résultats par la suite :).)

1 Estimation par intervalle

Dans cette première partie du devoir, vous allez tenter d'estimer un intervalle de valeurs possibles pour un paramètre que l'on cherche à estimer. Pour rappel, vous allez ainsi pouvoir quantifier l'incertitude de notre estimateur, au lieu de produire une seule estimation comme lors du premier devoir.

Vous allez étudier la magnitude apparente des étoiles $Vmag$ dans cette première section. En particulier, il vous est demandé de construire un intervalle de confiance au niveau de confiance $1 - \alpha = 0.95$ pour la moyenne de cette variable et ce, de deux manières différentes.

1.1 Approximation Normale

Vous allez ici construire un intervalle de confiance *approximé*. En effet, il vous est demandé de faire une approximation Normale, *i.e.* de considérer que la distribution de l'estimateur considéré est approximativement Normale.

Sans hypothèse quant à la distribution de $Vmag$ dans la population, une telle approximation ne peut cependant être faite qu'à certaine(s) condition(s). On suppose les données d'échantillons indépendantes et identiquement distribuées.

Quelle(s) condition(s) doive(nt) être remplie(s) ? Quel théorème cette(ces) condition(s) permet(tent)-elle(s) d'appliquer ?

Pour pouvoir construire un intervalle de confiance approximé, deux conditions doivent être satisfaites :

1. La taille de l'échantillon (n) doit être suffisamment grande. En pratique, on considère cette approximation comme **acceptable** à partir de n 30.
2. Cette condition permet d'appliquer le **Théorème Central Limite** qui stipule que pour n $\to \infty$:

$$\frac{\hat{\theta} - \theta}{\mathrm{SE}(\hat{\theta})} \to \mathcal{N}(0,1) \text{ en distribution pour toute valeur de } \theta$$

Ainsi, plus la valeur de n est grande, meilleure sera l'approximation normale (peut importe la distribution d'origine grâce au TCL)

Affichez la taille de votre échantillon.

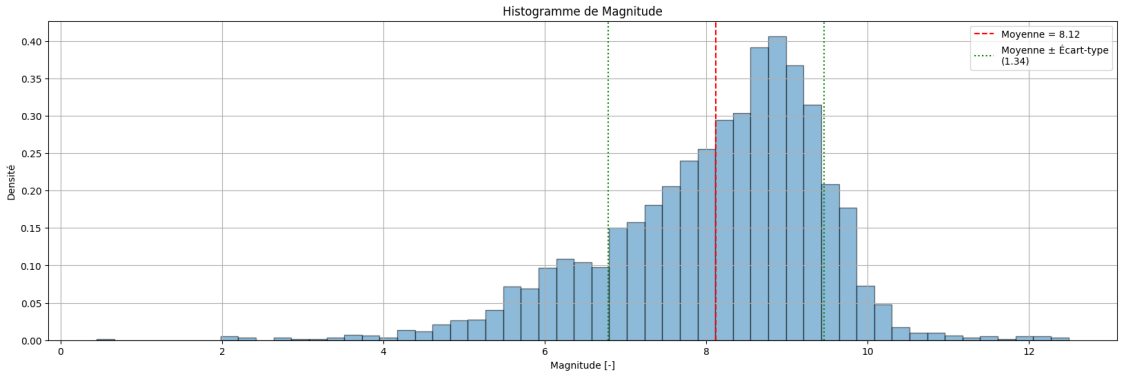
[5]:
```
n = len(data)
print("Taille de l'échantillon :", n)
```

```
Taille de l'échantillon : 3642
```

Ensuite, générez l'histogramme de cette variable dans l'échantillon.

[6]:
```
# Récupération des données
vmag = data["Vmag"]

# Calcul des statistiques
mean = np.mean(vmag)
std = np.std(vmag,  ddof=1) #ddof = 1 pour avoir (n-1) au dénominateur

# Créer la figure
fig, axs = plt.subplots(figsize=(20, 6))

# Histogramme
axs.hist(vmag, bins='auto', density=True, edgecolor='black', alpha=0.5)
axs.axvline(x=mean, color='red', linestyle='--', label=f'Moyenne = {mean:.2f}')
axs.axvline(x=mean + std, color='green', linestyle=':', label=f'Moyenne ±␣
 ↪Écart-type\n({std:.2f})')
axs.axvline(x=mean - std, color='green', linestyle=':')
axs.set_title('Histogramme de Magnitude')
axs.set_xlabel('Magnitude [-]')
axs.set_ylabel('Densité')
axs.grid(True)
axs.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x71f470aa4dc0>
```



Au vu de l'histogramme généré à partir des données récoltées, la condition précédemment énoncée pour l'hypothèse Normale était-elle indispensable ? Expliquez.

> Pour n = 3642, l'histogramme présente une forme relativement proche d'une Gaussienne, mais la répartition autour de la moyenne n'est pas celle d'une Gaussienne. La condition de n suffisamment grand est donc indispensable pour l'hypothèse Normale.

Calculez à présent l'intervalle de confiance pour la magnitude apparente moyenne au niveau de confiance demandé pour votre échantillon.

[7]:
```
# Niveau de confiance demandé
confidence = 0.95

# Conversion des données en numpy array
vmag = np.asarray(vmag)

# Calcul des bornes de l'intervalle de confiance
(borne_inf, borne_sup) = get_normal_CI(vmag, confidence)

print(borne_inf, borne_sup)
```

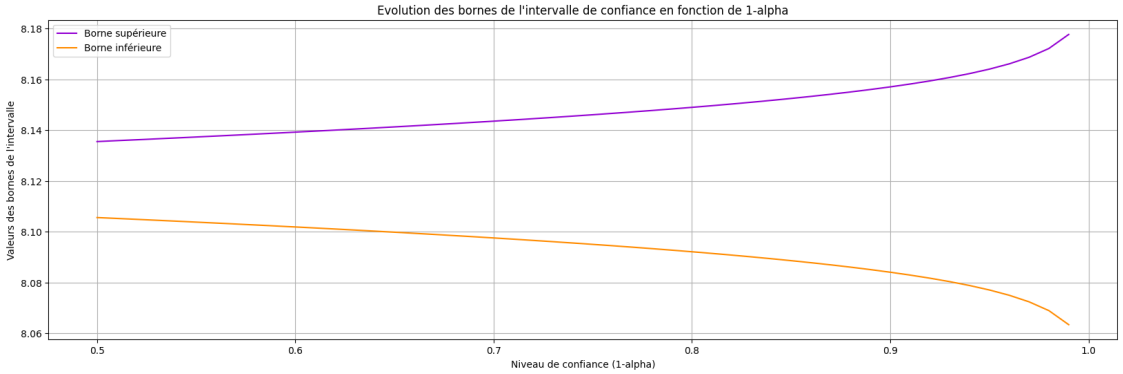
```
8.07704705185984 8.16401280536147
```

Calculez maintenant des intervalles de confiance pour la magnitude apparente moyenne aux niveaux $1-\alpha \in [0.50, 0.51, 0.52, ... 0.97, 0.98, 0.99]$. Représentez l'évolution des bornes de ces intervalles en fonction de $1-\alpha$ sur un graphe.

[8]:
```
confidence = np.arange(0.5, 1.0, 0.01)
bornes_inf = []
bornes_sup = []

# Calcul des bornes de l'intervalle de confiance pour les différents niveaux de␣
 ↪confiance
for i in range(len(confidence)):
  (borne_inf, borne_sup) = get_normal_CI(vmag, confidence)
  bornes_inf.append(borne_inf)
```

```
  bornes_sup.append(borne_sup)

# Figure
fig, axs = plt.subplots(figsize=(20, 6))
axs.plot(confidence, borne_sup, color='darkviolet', label="Borne supérieure")
axs.plot(confidence, borne_inf, color='darkorange', label="Borne inférieure")
axs.set_title("Evolution des bornes de l'intervalle de confiance en fonction de␣
 ↪1-alpha")
axs.set_xlabel("Niveau de confiance (1-alpha)")
axs.set_ylabel("Valeurs des bornes de l'intervalle")
axs.grid(True)
axs.legend()
```

[8]: <matplotlib.legend.Legend at 0x71f46e919ed0>



Qu'observez-vous lorsque l'on augmente (resp. diminue) le niveau de confiance $1-\alpha$? Interprétez.

> Quand on augmente le niveau de confiance $1-\alpha$, la valeur de la borne supérieure de l'intervalle augmente tandis que la valeur de la borne inférieure de l'intervalle diminue. L'intervalle de confiance s'élargit donc avec le niveau de confiance. En effet, lorsqu'on augmente le niveau de confiance, on augmente la probabilité que l'intervalle de confiance contienne la vraie valeur du paramètre estimé, et la largeur de l'intervalle augmente. (exemple: passer d'un niveau de confiance de 90% à 95% élargit l'intervalle mais offre une plus grande certitude.)
>
> C'est l'inverse qui se produit lorsqu'on diminue le niveau de confiance $1-\alpha$: la largeur de l'intervalle de confiance diminue. En effet, on diminue la probabilité que l'intervalle de confiance contienne la vraie valeur du paramètre estimé lorsque les bornes de l'intervalle se rapprochent (compromis à faire entre la précision de l'estimation (largeur de l'intervalle) et le niveau de confiance souhaité).

1.2 Bootstrapping

Vous allez à présent construire un intervalle en utilisant une autre méthode que la précédente : le *bootstrapping*.

Premièrement, de manière générique, donnez *toutes* les différentes étapes de la méthode du *bootstrapping* pour construire un intervalle de confiance au niveau de confiance $1-\alpha$ pour un paramètre θ à partir d'un échantillon de taille n, avec un nombre d'échantillons bootstrap m, et en utilisant la méthode du percentile.

Soit $\hat{\theta}$ l'estimateur de θ. Les différentes étapes de la méthode du *bootstrapping* sont :

1. Générer m échantillons bootstrap de taille n (avec m grand) en effectuant m tirages aléatoires avec remplacement à partir de l'échantillon.
2. Calculer l'estimateur $\hat{\theta}^*$ pour chaque échantillons bootstrap en utilisant la même procédure que pour calculer $\hat{\theta}$ sur les données réelles. On obtient ainsi m valeurs indépendantes $\hat{\theta}^*_1, \hat{\theta}^*_2, ..., \hat{\theta}^*_m$
3. Trier ces m valeurs dans l'ordre croissant pour obtenir la distribution bootstrap de $\hat{\theta}$.
4. Calculer directement les quantiles d'échantillon de la distribution bootstrap pour obtenir un intervalle au niveau de confiance $1-\alpha$. Soit $\beta = \frac{\alpha}{2}$, l'intervalle de confiance est donné par

$$[\hat{\theta}^*_{[\beta]}, \hat{\theta}^*_{[1-\beta]}]$$

Calculez un tel intervalle au niveau de confiance précisé plus tôt pour la magnitude apparente moyenne, avec un nombre d'échantillons bootstrap $m = 1000$.

[9]:
```
# Niveau de confiance demandé
confidence = 0.95

# Nombre d'échantillons bootstrap
m = 1000

# Récupération des bornes de l'intervalle et de la distribution
borne_inf, borne_sup, boot_dis = get_bootstrap(vmag, confidence, m)

print(borne_inf, borne_sup)
```

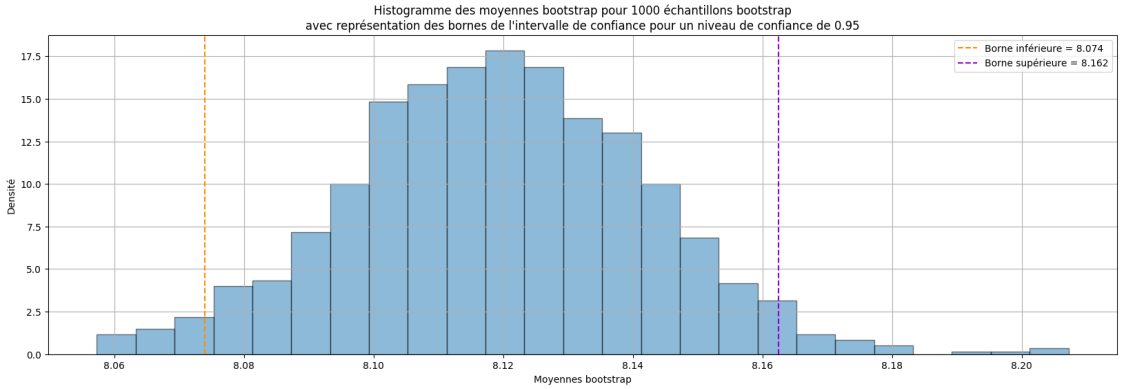
```
8.073952635914333 8.162492311916528
```

Générez l'histogramme des valeurs de l'estimateur pour vos échantillons bootstrap. Superposez à cet histogramme deux droites verticales afin de représenter les bornes de l'intervalle construit ci-avant.

[10]:
```
# Histogramme
fig, axs = plt.subplots(figsize=(20, 6))
axs.hist(boot_dis, bins='auto', density=True, edgecolor='black', alpha=0.5)
axs.axvline(x=borne_inf, color='darkorange', linestyle='--', label=f'Borne␣
 ↪inférieure = {borne_inf:.3f}')
axs.axvline(x=borne_sup, color='darkviolet', linestyle='--', label=f'Borne␣
 ↪supérieure = {borne_sup:.3f}')
axs.set_title(f"Histogramme des moyennes bootstrap pour {m} échantillons␣
 ↪bootstrap\navec représentation des bornes de l'intervalle de confiance pour␣
 ↪un niveau de confiance de {confidence}")
```

```
axs.set_xlabel('Moyennes bootstrap')
axs.set_ylabel('Densité')
axs.grid(True)
axs.legend()
```

[10]: <matplotlib.legend.Legend at 0x71f46e9e42b0>



2 Régression linéaire, qualité de l'estimation, résidus de régression et leverage score

Vous allez désormais étudier le concept de régression linéaire prédictive avec les données de votre échantillon. Lorsqu'il vous est demandé d'estimer les paramètres d'un modèle de régression, vous pouvez utiliser la librairie *scikit-learn*.

Pour rappel, un modèle de regression linéaire peut être écrit sous la forme

$$E(Z \mid \mathbf{X} = (x_1, \dots, x_k), (\theta_0, \theta_1, \dots \theta_k)) = \mu(\mathbf{x} \mid (\theta_0, \theta_1, \dots \theta_k)) = \theta_0 + \theta_1 x_1 + \dots + \theta_k x_k,$$

avec Z la variable aléatoire d'observation, $\mathbf{X}$ le vecteur $(X_1, \dots, X_k)$ de variables aléatoires prédictives et $(\theta_0, \theta_1, \dots \theta_k)$ les paramètres ou coefficients de régression (avec θ_0 l'intercept).

Un ami astronome vous suggère de commencer par prédire la différence entre la magnitude absolue et la magnitude apparente (cette différence devient ainsi votre variable d'observation).

Dans un premier temps, tentez de faire un modèle simple de régression linéaire en ne considérant uniquement que la variable prédictive : *l'angle de la parallaxe*. Ajustez également l'intercept. Affichez les coefficients de régression.

[11]:
```
# Variable prédictive : angle de la parallaxe
plx = data["Plx"].values.reshape(-1, 1)

# Variable d'observation : différence entre la magnitude absolue et la␣
  ↪magnitude apparente
amag = data["Amag"].values
vmag = data["Vmag"].values
```

```
diff = amag - vmag

# Modèle choisi
model = LinearRegression(fit_intercept=True)

# Ajustement du modèle
model.fit(plx, diff)

# Affichage des coefficients de régression
(intercept, coef) = (model.intercept_, model.coef_)

print("Intercept :", intercept)
print("Coefficient :", coef)
```

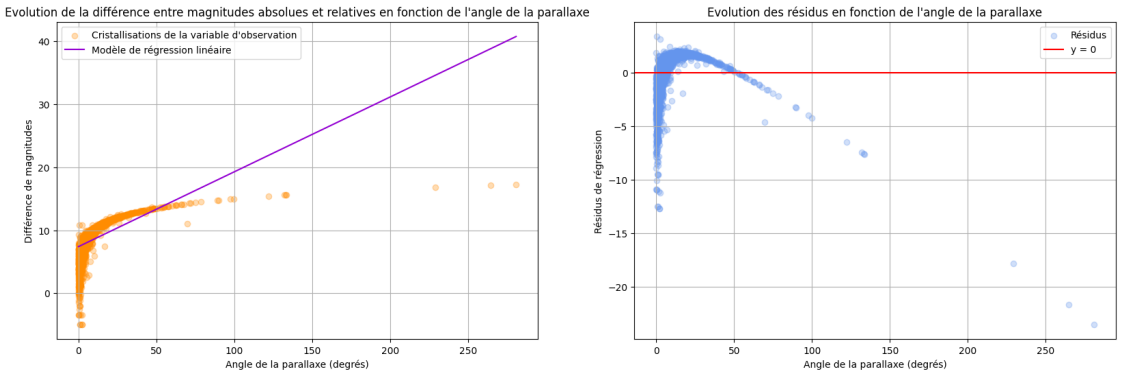
```
Intercept : 7.390914193075679
Coefficient : [0.11882462]
```

Générez un nuage de point représentant les cristallisations de votre variable d'observation en fonction de votre variable prédictive. Sur le même graphique, représentez votre modèle de régression linéaire. Sur un autre graphique, montrez l'évolution de vos résidus par rapport à la variable prédictive.

[12]:
```
fig, axs = plt.subplots(1, 2, figsize=(20, 6))

# Nuage de points
axs[0].scatter(plx, diff, color='darkorange', alpha = 0.3,␣
 ↪label="Cristallisations de la variable d'observation")

# Modèle de régression linéaire
x = np.linspace(min(plx), max(plx), 100)
y = model.predict(x.reshape(-1, 1))

axs[0].plot(x, y, color='darkviolet', label="Modèle de régression linéaire")
axs[0].set_title("Evolution de la différence entre magnitudes absolues et␣
 ↪relatives en fonction de l'angle de la parallaxe")
axs[0].set_xlabel("Angle de la parallaxe (degrés)")
axs[0].set_ylabel("Différence de magnitudes")
axs[0].grid(True)
axs[0].legend()

# Evolution des résidus par rapport à la variable prédictive
predictions = model.predict(plx.reshape(-1, 1))
residus = diff - predictions

axs[1].scatter(plx, residus, color='cornflowerblue', alpha = 0.3,␣
 ↪label="Résidus")
axs[1].axhline(y = 0, color='red', label="y = 0")
```

```
axs[1].set_title("Evolution des résidus en fonction de l'angle de la parallaxe")
axs[1].set_xlabel("Angle de la parallaxe (degrés)")
axs[1].set_ylabel("Résidus de régression")
axs[1].grid(True)
axs[1].legend()
```

[12]: <matplotlib.legend.Legend at 0x71f46e8db010>



Qu'observez-vous ? Le modèle a-t-il une capacité de représentation suffisante (c'est sa capacité à capturer la relation complexe entre la variable d'observation et la variable prédictive) ? Si non, quelles améliorations peuvent être introduites au vu de l'évolution des résidus ?

> La superposition du modèle et du nuage de points montre que le modèle ne suit pas la tendance générale des données, particulièrement pour les angles de parallaxe élevés. Le modèle n'a donc pas une capacité de représentation suffisante.
>
> La forme non linéaire des résidus et leur distribution non aléatoire autour de 0 nous montre que la relation entre la variable prédictive et la variable d'observation est très certainement non linéaire. Le modèle pourrait donc être amélioré soit en utilisant un modèle de régression non linéaire, soit en appliquant une transformation non linéaire (log, exp) à une des deux variables.

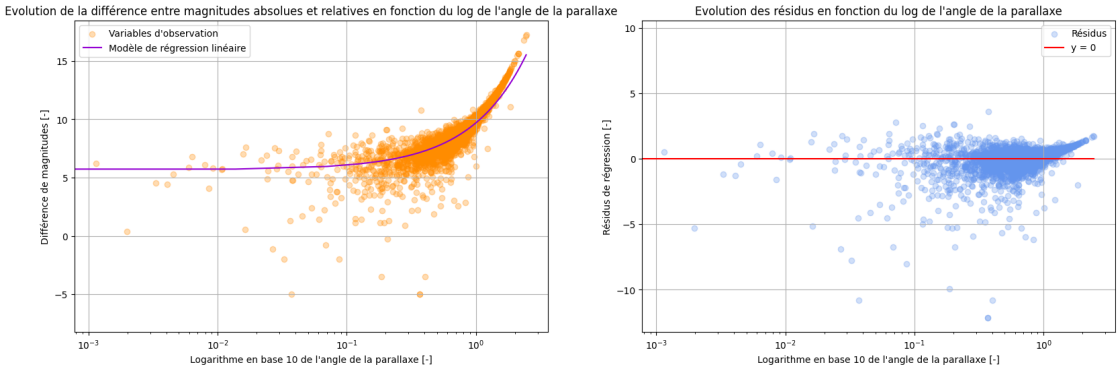
Désormais un de vos amis vous conseille de travailler avec le $\log_{10}(Plx)$ à la place de Plx. On restera dans ce cadre jusqu'à la fin de la section sur la régression linéaire.

Générez un nuage de points représentant les valeurs de votre variable d'observation en fonction de votre nouvelle variable prédictive. Sur le même graphique, tracez le modèle de régression linéaire que vous venez d'obtenir avec cette nouvelle procédure. Comme précédemment, affichez également les résidus sur un autre graphe. Enfin, calculez et affichez le score R^2 ainsi que la MSE du modèle en utilisant les données ayant servi à l'ajustement.

[13]:
```
# Nouvelle variable prédictive : log10(angle de la parallaxe)
log10plx = np.log10(plx).reshape(-1, 1)

fig, axs = plt.subplots(1, 2, figsize=(20, 6))
```

```
# Nuage de points
axs[0].scatter(log10plx, diff, color='darkorange', alpha = 0.3,␣
 ↪label="Variables d'observation")

# Modèle de régression linéaire
log10model = LinearRegression(fit_intercept=True)
log10model.fit(log10plx, diff)
log10x = np.linspace(min(log10plx), max(log10plx), 100)
y = log10model.predict(log10x)

axs[0].set_xscale("log", base = 10)
axs[0].plot(log10x, y, color='darkviolet', label="Modèle de régression␣
 ↪linéaire")
axs[0].set_title("Evolution de la différence entre magnitudes absolues et␣
 ↪relatives en fonction du log de l'angle de la parallaxe")
axs[0].set_xlabel("Logarithme en base 10 de l'angle de la parallaxe [-]")
axs[0].set_ylabel("Différence de magnitudes [-]")
axs[0].grid(True)
axs[0].legend()

# Evolution des résidus par rapport à la variable prédictive
log10predictions = log10model.predict(log10plx)
residuals = diff - log10predictions
yref = 0 * log10x

axs[1].set_xscale("log", base = 10)
axs[1].scatter(log10plx.ravel(), residuals, color='cornflowerblue', alpha = 0.
 ↪3, label="Résidus")
axs[1].plot(log10x, yref, color='red', label="y = 0")
axs[1].set_title("Evolution des résidus en fonction du log de l'angle de la␣
 ↪parallaxe")
axs[1].set_xlabel("Logarithme en base 10 de l'angle de la parallaxe [-]")
axs[1].set_ylabel("Résidus de régression [-]")
axs[1].grid(True)
axs[1].legend()

# Calcul du coefficient de détermination R²
r2 = log10model.score(log10plx, diff)
print("Coefficient de détermination R² :", r2)

# Calcul de la MSE
#mse = mean_squared_error(diff, predictions)
mse = mean_squared_error(diff, log10predictions)
print("MSE :", mse)
```

```
Coefficient de détermination R² : 0.713973974310058
MSE : 1.5760381767243705
```



Que pensez-vous de ce modèle ? Le protocole de calcul du coefficient R^2 et de la MSE fournit-il une mesure fiable de la qualité du modèle lorsque ce dernier est appliqué à des variables prédictives nouvellement cristallisées, qui n'ont pas été utilisées lors de l'ajustement ?

> Ce modèle semble meilleur que le modèle précédent : la courbe obtenue suit bien la tendance générale des données avec cependant un bémol lorsque la valeur de l'angle de la parallaxe devient supérieure à 5 degrès (log10(Plx) > 0): la courbe se situe constamment en-dessous de la tendance générale des données. De plus les résidus sont plus uniformément distribués autour de zéro.
>
> Si R^2 et la MSE sont calculés à partir de nouvelles observations qui n'ont pas été utilisées lors de l'ajustement, ils peuvent donner des indications sur la qualité du modèle, mais ne constituent pas nécessairement une mesure fiable. En effet, R^2 comme MSE mesurent tous les deux dans quelle mesure le modèle prédit les observations en fonction des variables prédictives, mais leur pertinence dépend de la représentativité des nouvelles données. R^2 et MSE ne sont donc pertinents que s'ils sont calculés sur des données différentes de celles qui ont servi pour la formation du modèle, sans pour autant garantir sa qualité.

Commentez le graphe des résidus. Une hypothèse de la régression linéaire est-elle rompue ? (Vous pouvez aussi tenter d'expliquer qualitativement pourquoi les résidus suivent cette forme.)

> La forme "évasée" du nuage de points des résidus montre qu'il y a une erreur de prédiction non négligeable pour les faibles valeurs d'angle de la parallaxe. Cette erreur de prédiction diminue lorsque la valeur de l'angle de la parallaxe augmente.
>
> On peut également remarquer que presque toutes les valeurs de résidus sont positives au-delà d'une valeur d'angle de parallaxe d'environ 10 degrés, ce qui est logique étant donné que le modèle se situe constamment en-dessous de la tendance générale des données pour les plus grandes valeurs d'angle de la parallaxe.
>
> L'hypothèse rompue de la régression linéaire est celle **d'holiscédasticité** (disant que la variance des résidus est constante peut importe la variable prédictive). En effet, ici la variance des résidus change selon la valeur de $log10(Plx)$.

Il vous est désormais demandé d'étudier le leverage score des différentes cristallisations.

Une manière d'obtenir le leverage score de la $j^{\text{ème}}$ observation $\mathbf{x}_j = (x_{j1}, x_{j2}, ..., x_{jK})$ est de revenir

aux équations Normales de la régression.

Dans le cadre de la régression linéaire, il existe une transformation linéaire reliant les cristallisations des variables d'observation aux valeurs prédites par le modèle. Plus précisément, on note :

- $\mathbf{X}$: la matrice contenant les cristallisations des variables prédictives

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1K} \\ 1 & x_{21} & \cdots & x_{2K} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \cdots & x_{nK} \end{pmatrix},$$

- $\mathbf{y} = (y_1, \ldots, y_n)^T$: le vecteur des cristallisations des variables d'observation,
- n : le nombre total de cristallisations.

Il est possible de construire un modèle de régression linéaire ajusté sur ces données. On peut alors montrer que le vecteur des prédictions $\hat{\mathbf{y}}$ s'exprime de la manière suivante :

$$\hat{\mathbf{y}} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{H}\mathbf{y}$$

où $\mathbf{H}$ est appelée la Hat matrix car elle "ajoute le chapeau" à $\mathbf{y}$.

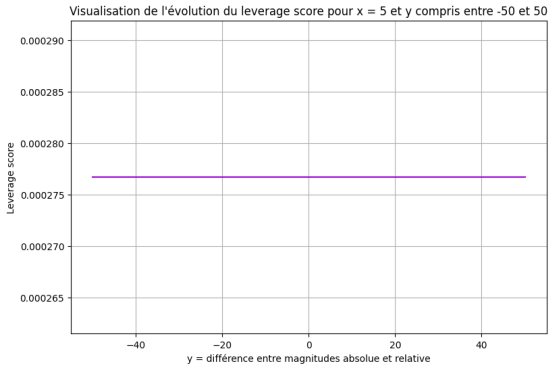
Il est possible de trouver le leverage score de la $j^{\text{ème}}$ observation $\mathbf{x}_j$ en prenant le $j$ème élément dans la diagonale de $\mathbf{H}$.

Nous allons maintenant essayer de comprendre de manière plus intuitive ce que représente ce leverage score.

Observez l'évolution du leverage score et du résidu associés à un point que vous ajoutez à votre dataset, en fonction de la position de ce dernier. Initialement situé en $(x_1, y) = (5, -50)$, le point doit se déplacer en ligne droite jusqu'à $(5, 50)$. Visualisez l'évolution de ces deux valeurs au cours de ce déplacement. (N'oubliez pas d'ajuster le modèle lorsque chaque point est ajouté)

[20]:
```
# Point à ajouter au dataset
# Angle de la parallaxe fixe : x = 5 degrés (log10 car on travaille avec␣
 ↪log10(PLx))
x = np.log10(5)

# Différence de magnitude : variation entre -50 et 50
y = np.linspace(-50, 50, 101)
length_y = y.size

leverages = []
residus = []

# Ajout du point au dataset et calcul du leverage score et du résidu associé à␣
 ↪ce point
for i in range(length_y):
    # La valeur de x reste constante donc le vecteur x_new également
    x_new = np.append(log10plx, x)

    # Ajout du point au dataset
```

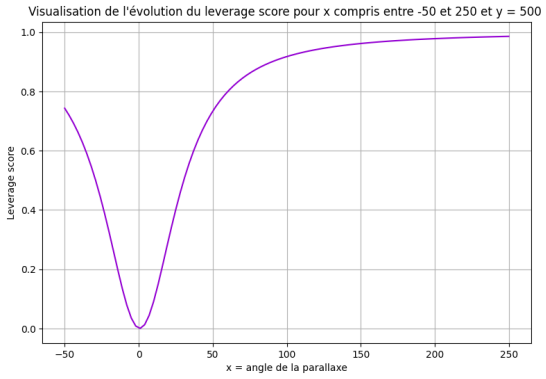
```
    y_new = np.append(diff, y[i])

    # Calcul de la Hat matrix pour ce point
    w = np.ones(len(x_new))
    X_transpose = np.stack((w, x_new))
    X = X_transpose.T
    X_inv = np.linalg.inv(X_transpose @ X)
    H = X @ X_inv @ X_transpose

    # Récupération du leverage score associé au point ajouté
    leverages.append(H[-1, -1])

    # Calcul du résidu associé au point ajouté au dataset
    model = LinearRegression(fit_intercept=True)
    model.fit(x_new.reshape(-1, 1), y_new)
    predictions_new = model.predict(x_new.reshape(-1, 1))
    res = y_new[-1] - predictions_new[-1]
    residus.append(res)

# Visualisation de l'évolution des valeurs de leverage score et de résidu
fig, axs = plt.subplots(1, 2, figsize=(20, 6))

# Leverage scores
axs[0].plot(y, leverages, color='darkviolet')
axs[0].set_xlabel('y = différence entre magnitudes absolue et relative')
axs[0].set_ylabel('Leverage score')
axs[0].set_title("Visualisation de l'évolution du leverage score pour x = 5 et␣
 ↪y compris entre -50 et 50")
axs[0].grid(True)

# Résidus
axs[1].plot(y, residus, color='darkorange', label="Evolution des résidus")
axs[1].set_xlabel('y = différence entre magnitudes absolue et relative')
axs[1].set_ylabel('Résidu')
axs[1].set_title("Visualisation de l'évolution du résidu pour x = 5 et y␣
 ↪compris entre -50 et 50")
axs[1].grid(True)
```

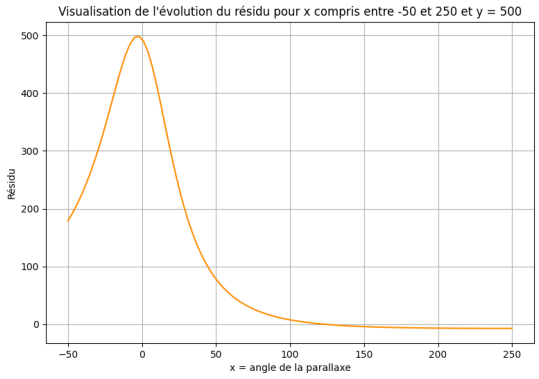


Faites la même chose qu'au point précédent mais avec un point se déplaçant entre $(-50, 500)$ et $(250, 500)$.

[27]:
```
# Point à ajouter au dataset
# Angle de la parallaxe : variation entre -50 et 250 degrés
x = np.linspace(-50, 250, 101)

# Différence de magnitude fixe : y = 500
y = 500
length_x = x.size

leverages = []
residus = []

# Ajout du point au dataset et calcul du leverage score et du résidu associé à␣
 ↪ce point
for i in range(length_x):

  # Ajout du point au dataset
  x_new = np.append(log10plx, x[i])

  y_new = np.append(diff, y)

  # Calcul de la Hat matrix H
  w = np.ones(len(x_new))
  X_transpose = np.stack((w, x_new))
  X = X_transpose.T
  X_inv = np.linalg.inv(X_transpose @ X)
  H = X @ X_inv @ X_transpose

  # Récupération du leverage score du point ajouté au dataset
  leverage = H[-1, -1]
  leverages.append(leverage)
```

```
  # Calcul du résidu associé au point ajouté au dataset
  model = LinearRegression(fit_intercept=True)
  model.fit(x_new.reshape(-1, 1), y_new)
  predictions_new = model.predict(x_new.reshape(-1, 1))
  res = y_new[-1] - predictions_new[-1]
  residus.append(res)

# Visualisation de l'évolution des valeurs de leverage score et de résidu
fig, axs = plt.subplots(1, 2, figsize=(20, 6))

# Leverage scores
axs[0].plot(x, leverages, color='darkviolet')
axs[0].set_xlabel('x = angle de la parallaxe')
axs[0].set_ylabel('Leverage score')
axs[0].set_title("Visualisation de l'évolution du leverage score pour x compris␣
 ↪entre -50 et 250 et y = 500 ")
axs[0].grid(True)

# Résidus
axs[1].plot(x, residus, color='darkorange')
axs[1].set_xlabel('x = angle de la parallaxe')
axs[1].set_ylabel('Résidu')
axs[1].set_title("Visualisation de l'évolution du résidu pour x compris entre␣
 ↪-50 et 250 et y = 500")
axs[1].grid(True)
```





Qu'observez-vous aux deux points précédents ? En utilisant ces résultats, expliquez à quoi correspond la notion de leverage ainsi que sa possible utilité. Vous pouvez vous inspirez de diverses sources telles que le TP.

> Lorsque seule la valeur de y (différence entre magnitudes absolues et relatives) varie, la Hat matrice demeure la même pour les différentes valeurs de y puisque x ne change pas. Par conséquent, le leverage score associé au point ajouté au dataset reste inchangé et proche de zéro. En revanche, la valeur de résidu associée au point ajouté varie

linéarement avec la valeur de y, ce qui est logique puisque le résidu est la différence entre la valeur observée et la valeur prédite.

En revanche, lorsque seule la valeur de x (angle de la parallaxe) varie pour une valeur de y fixée à une très grande valeur de 500, leverage score et résidu évoluent de manière totalement opposée :

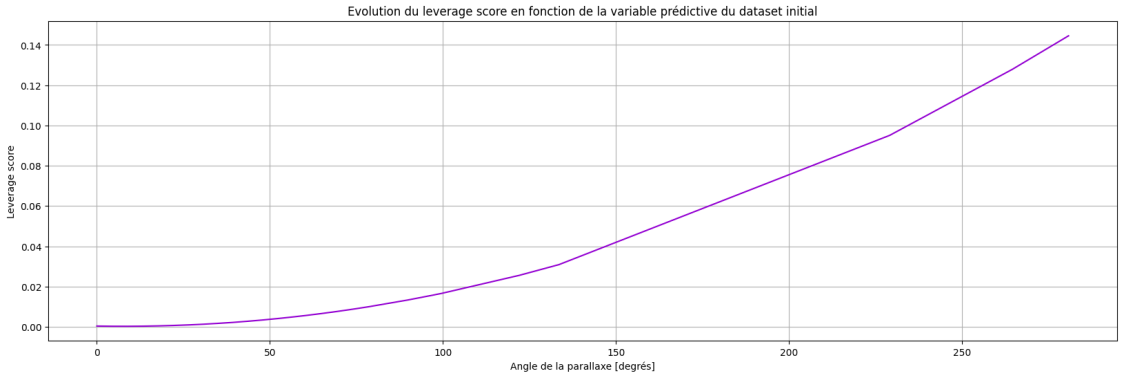
- Pour x compris entre -50 et 0 : le leverage score diminue depuis une valeur élevée (0.743) pour x = -50 jusqu'à zéro pour x = 0, tandis que le résidu augmente jusqu'à atteindre une valeur de 500 pour x = -2 avant de diminuer légèrement;
- Pour x compris entre 0 et 250 : le leverage score augmente rapidement jusqu'à retourner à une valeur de 0.743 pour x = 50, puis continue à augmenter asympototiquement vers la valeur 1 à mesure que x augmente, tandis que le résidu connaît une décroissance rapide pour des angles compros entre 0 et 100 degrès, jusqu'à une valeur proche de zéro pour les angles légérement supérieurs à 100 degrés et jusqu'à 250 degrés.

Le leverage score est une mesure de la distance entre la valeurs associée à une observation et celles des autres observations : plus le leverage score d'une observation est élevée, plus cette observation est éloignée des autres observations. Les données présentant un grand leverage score sont donc fort distantes, et peuvent donc être considérées comme des outliers par rapport aux variables prédictives, et vont avoir une grande influence sur le résidu de régression.

Observez l'évolution du leverage score en fonction de la variable prédictive du dataset initial.

[39]:
```
# Matrice X : tri sur les valeurs de plx, variable prédictive du dataset initial
plx_sort = np.sort(plx.ravel())
length_plx = len(plx.ravel())
w = np.ones(length_plx)

# Conversion en arrays numpy
plx_sort = np.array(plx_sort)
w = np.array(w)

# Construction de la matrice X
X_transpose = np.vstack((w, plx_sort))  # vstack au lieu de stack

# Calcul de la Hat matrix H
X = X_transpose.T
X_inv = np.linalg.inv(X_transpose @ X)
H = X @ X_inv @ X_transpose
diag_H = np.diag(H)

# Figure
fig, axs = plt.subplots(figsize=(20, 6))
axs.plot(plx_sort, diag_H, color='darkviolet')
axs.set_xlabel('Angle de la parallaxe [degrés]')
axs.set_ylabel('Leverage score')
```

```
axs.set_title("Evolution du leverage score en fonction de la variable␣
 ↪prédictive du dataset initial")
axs.grid(True)
```



Que pensez-vous de la courbe ainsi obtenue ?

> On remarque que plus l'angle de la parallaxe est élevé, plus le leverage score est élevé, ce qui est cohérent avec les résulats obtenus précédemment : les valeurs d'angle de parallaxe très élevées correspondent à des points du dataset que l'on peut considérer comme des outliers.

3 Régression logistique

Nous allons désormais utiliser un modèle de régression logistique pour résoudre un problème de classification binaire : il nous faut prédire si l'étoile est une naine ou une géante. Ceci va se faire en utilisant deux variables prédictives : - l'indice de couleur B-v, - la magnitude absolue $Amag$.

Vous pouvez utiliser la classe LogisticRegression de scikit-learn pour cette partie du devoir.

Pour rappel, un modèle de régression logistique estime la probabilité qu'une observation appartienne à une catégorie en appliquant une fonction sigmoïde sur le résultat d'une régression linéaire

$$P(Y = 1 \mid \mathbf{X} = \mathbf{x};\ ) = \frac{\exp(\lambda(\mathbf{x}))}{1 + \exp(\lambda(\mathbf{x}))}$$

où

$$\lambda(\mathbf{x}) = \theta_0 + \theta_1 x_1 + ... + \theta_K x_K.$$

Dans la formule ci-dessus: - x_k est la cristallisation de la $k^{\text{ème}}$ variable prédictive, - θ_0 est l'intercept, - θ_k est le $k^{\text{ème}}$ coefficient de la régression.

Pour prédire une classe spécifique, il suffit de définir un seuil (threshold) sur les probabilités. Dans scikit-learn, ce seuil est fixé à 0.5 par défaut. Plus précisément, dès que la probabilité prédite est supérieure à 0.5, l'observation est associée à la classe $Y = 1$ (l'étoile géante dans ce cas). Sinon, elle est associée à la classe $Y = 0$ (l'étoile naine).

NB : Bien que cela ne soit pas requis dans cet exercice, il est courant en pratique d'ajuster ce seuil en fonction des spécificités du contexte. Par exemple, dans les applications biomédicales, les conséquences d'une erreur peuvent être coûteuses : détecter un cancer chez une personne en parfaite santé, par exemple. Dans de tels cas, il est pertinent de modifier le seuil pour minimiser le risque d'erreurs critiques.

Ajustez un modèle de régression logistique pour déterminer si une étoile est une naine ou une géante, en utilisant l'indice de couleur et la magnitude absolue. Évaluez ensuite le pourcentage de prédictions correctes (également appelé accuracy) sur les données utilisées pour l'ajustement.

[26]:
```
# Variables prédictives : (indice de couleur B-V, magnitude absolue Amag)
BV = data["B-V"]

BV = np.asarray(BV)
amag = np.asarray(amag)
X_transpose = np.stack((BV, amag))
X = X_transpose.T

# Variable d'observation : classe de l'étoile (naine ou géante)
var = "TargetClass"
Y = data[var]
Y = np.asarray(Y)

# Régression logistique
model = LogisticRegression()
model.fit(X, Y)

# Calcul du pourcentage de prédictions correctes (accuracy)
predictions = model.predict(X)
accuracy = model.score(X, Y)
accuracy = accuracy * 100
print("Accuracy :", accuracy, "%")
```

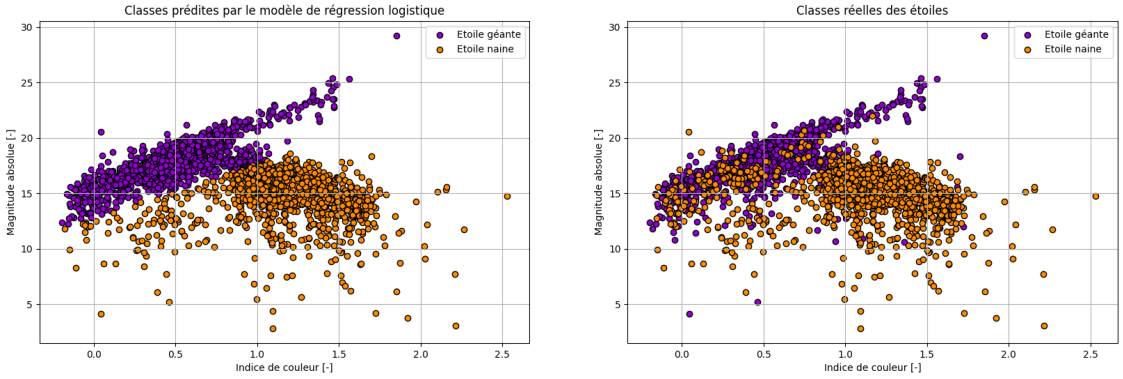
```
Accuracy : 89.48380010982976 %
```

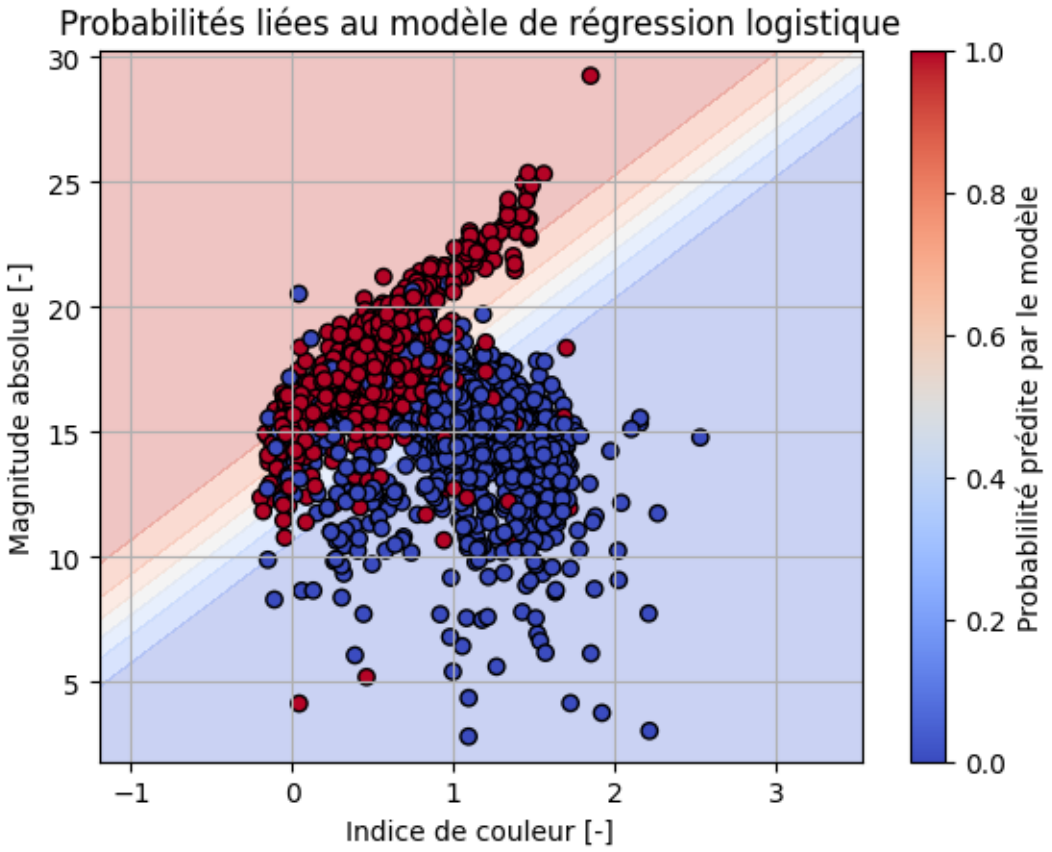
Affichez un nuage de points représentant les deux variables prédictives, avec une couleur indiquant la classe prédite par le modèle. Réalisez ensuite un second graphique où la couleur correspond à la classe réelle. Il vous est également demandé d'afficher les probabilités liées à ce modèle en utilisant *DecisionBoundaryDisplay* sur un autre graphe.

[ ]:
```
# Classe prédite par le modèle
BV_naine = []
amag_naine = []
BV_geante = []
amag_geante = []

length = predictions.size
for i in range(length):
  if predictions[i] == 1:
```

```
    BV_geante.append(BV[i])
    amag_geante.append(amag[i])
  else:
    BV_naine.append(BV[i])
    amag_naine.append(amag[i])

# Représentation des classes prédites par le modèle
fig, axs = plt.subplots(1, 2, figsize=(20, 6))
axs[0].scatter(BV_geante, amag_geante, color='darkviolet', edgecolors='k',␣
 ↪marker='o', label="Etoile géante")
axs[0].scatter(BV_naine, amag_naine, color='darkorange', edgecolors='k',␣
 ↪marker='o', label="Etoile naine")
axs[0].set_xlabel('Indice de couleur [-]')
axs[0].set_ylabel('Magnitude absolue [-]')
axs[0].set_title("Classes prédites par le modèle de régression logistique")
axs[0].grid(True)
axs[0].legend()

# Classes réelles
BV_naine = []
amag_naine = []
BV_geante = []
amag_geante = []

for i in range(length):
  if Y[i] == 1:
    BV_geante.append(BV[i])
    amag_geante.append(amag[i])
  else:
    BV_naine.append(BV[i])
    amag_naine.append(amag[i])

# Représentation des classes réelles des étoiles
axs[1].scatter(BV_geante, amag_geante, color='darkviolet', edgecolors='k',␣
 ↪marker='o', label="Etoile géante")
axs[1].scatter(BV_naine, amag_naine, color='darkorange', edgecolors='k',␣
 ↪marker='o', label="Etoile naine")
axs[1].set_xlabel('Indice de couleur [-]')
axs[1].set_ylabel('Magnitude absolue [-]')
axs[1].set_title("Classes réelles des étoiles")
axs[1].grid(True)
axs[1].legend()

# Représentation des probabilités liées au modèle de régression logistique
display = DecisionBoundaryDisplay.from_estimator(model, X,␣
 ↪response_method="predict_proba", cmap=plt.cm.coolwarm, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c = Y, edgecolor="k", cmap=plt.cm.coolwarm)
```

```
colorbar = plt.colorbar()
colorbar.set_label("Probablilité prédite par le modèle")
plt.xlabel('Indice de couleur [-]')
plt.ylabel('Magnitude absolue [-]')
plt.title("Probabilités liées au modèle de régression logistique")
plt.grid(True)
```





Analysez cela. Votre modèle vous semble-t-il bon ? Ce modèle souffre-t-il de limitations ? Comment peut-on l'améliorer ?

Le modèle est globalement bon : étoiles géantes et étoiles naines sont correctement classifiées dans la majorité des cas comme le montrent les 3 graphiques.

Il souffre néanmoins de limitations pour la classification des étoiles naines qui ont un indice de couleur inférieur à 1 et une magnitudes absolue comprise entre 10 et 20. Ces étoiles naines sont catégorisées comme étoiles géantes par le modèle.

En effet, on peut remarquer sur le graphique représentant les classes réelles des étoiles que, dans ces ranges de valeurs, les données ne sont pas séparables de manière linéaire : étoiles géantes et étoiles naines se retrouvent mélangées. C'est d'ailleurs bien visible sur le troisième graphe qui montre les probabilités liées au modèle de régression logistique : une bande intérmédiaire de probabilités comprises entre 0.2 et 0.8 séparent les 2 catégories.

Pour améliorer ce modèle, on pourrait ajuster le seuil de décisions pour mieux gérer la zone de chevauchement où les probabilités sont entre 0.2 et 0.8.