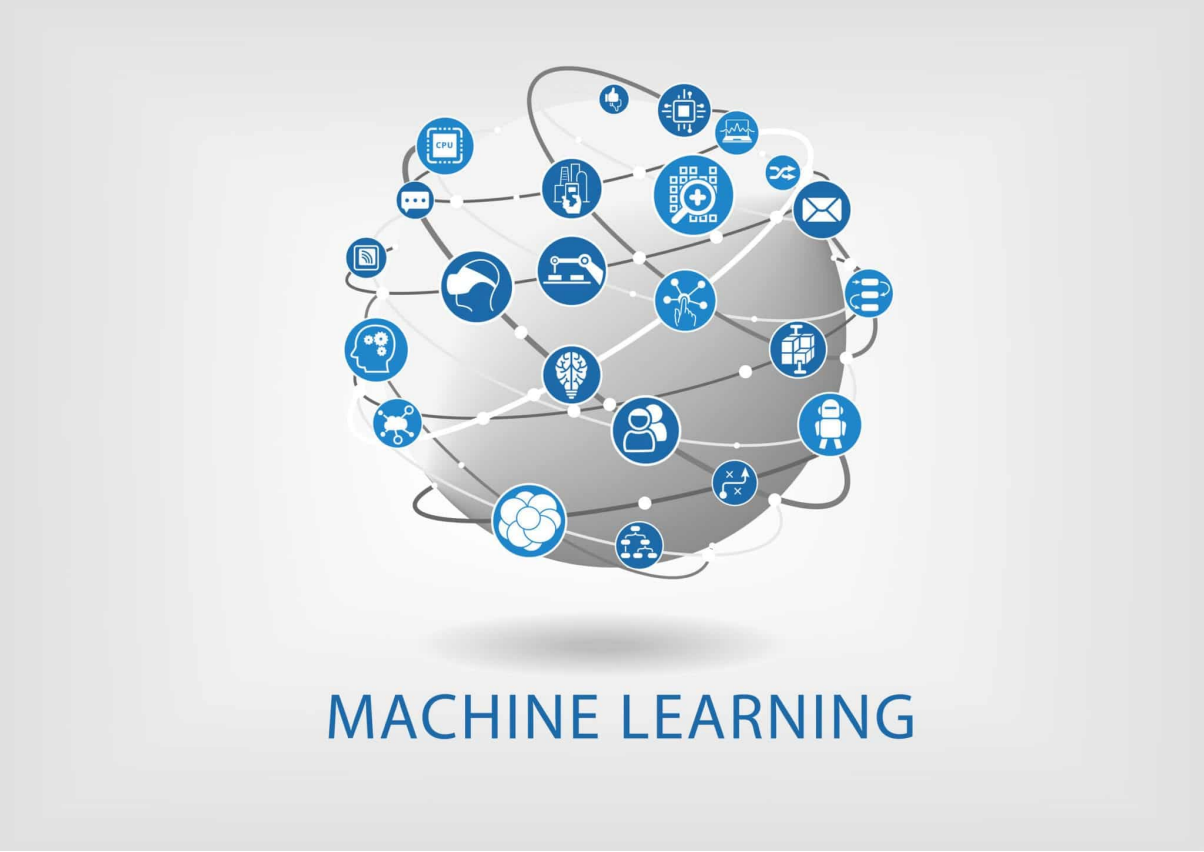
MACHINE LEARNING PROJECT 2024-2025

Classification de Tweet Financier



LEVILLAYER Noam, LAHJAJI Hamid, VIRELIZIER Aubin

Edité par Aubin VIRELIZIER

Table des matières

1 Introduction

Le **Machine Learning** ou apprentissage automatique est définit comme une branche de l'intelligence artificielle. Il permet à une machine d'apprendre à partir de données, sans être explicitement programmé pour chaque tâche.

Les premières apparitions de cette sciences ont lieu en **1950**, avec notamment la publication d"un article de **Alan Turning**, "Computing Machinery and Intelligence", où il discute du rôle de la machine et si celle-ci peut penser ? ?

Mais c'est surtout, dans les **années soixante** que le Machine Learning émerge, avec l'invention de réseau de neurone, par **Frank Rosenbelt**, et des premiers algorithmes supervisés, comme K-NN.

Dans les pages qui suivent nous tenterons de traiter une base de données d'environ 10 000 tweets financiers classés selon leurs sentiments. Les tweets peuvent être positifs, négatifs ou neutres.

L'objectif de ce projet est de chercher l'algorithme qui prédit le mieux les classes des tweets. Pour ce faire, nous analyserons 6 modèles binaires (négatif et non-négatif) et un modèle multiclasse (neutre, positif, négatif). Pour comparer ces modèles nous utiliserons, entre autre, les métriques suivantes : F1-score, Accuracy, et Rappel.

Enfin, les algorithmes reposeront en grande partie sur une vectorisation **Bag of Word**, qui sera expliquer dans une première partie. La présentation des modèles sera accompagnée d'une description sommaire de leur mécanisme avec des exemples concrets, puis, du code python utilisé pour le fonctionnement du modèle, et se finira par un tableau de résultat des métriques.

2 Pré-traitement des données

2.1 Nettoyage des données textuelles

La base de donné contient environ 9500 tweets classés de 0 à 2 selon leur sentiment (0=négatif, 1=neutre, 2= positif). C'est sur cette base de donné que les algorithmes vont s'entrainer pour apprendre et prédire le sentiment de nouveaux tweets.

Dans un premier temps, le but va être de nettoyer la base de donné. Celle-ci contient 2 colonnes nommées respectivement *text* et *label*. Nous allons surtout apporter, en premier lieu, des modifications à la colonne *text*. Nous utiliserons la librairie *nlkt* (*Natural language Toolkit*) qui va permettre de traiter les données textuelles avec des techniques de *Tokenisation*, *Stopwords*, *Stemming*, *Lemmatization*, et également la fonction *"regex"*.

Nous pouvons observer ces opérations à partir du code ci-dessous :

```
1 def clean_and_lemmatize(text):
2         text = text.lower()
3         text = re.sub(r'\$\w+', '', text) #supress actions/tickers
4         text = re.sub(r'http\S+|www.\S+', '', text) #supress url
5         text = re.sub(r'[^a-z\s]', '', text) #supress special character and number
6         tokens = nltk.word_tokenize(text)
7         tokens = [word for word in tokens if word not in stopwords.words('english')]
8         lemmas = [lemmatizer.lemmatize(token) for token in tokens]
9 return ' '.join(lemmas)
```

Nous retrouvons donc la majorité des éléments qui permette de nettoyer notre base de donné :

- Ligne 2 : nous mettons tout le texte en minuscule.
- Ligne 3 : nous utilisons la fonction *regex* pour supprimer les noms d'actions et tickers.
- Ligne 4 : nous supprimons les urls
- Ligne 5 : nous supprimons les caractères spéciaux tel que les "$", ",", ".", ":", "—" ect...
- Ligne 6, : nous découpons la phrase mot par mot (token by token)
- Ligne 7 : nous supprimons les *stopword*, c'est à dire les mots en anglais qui n'apportent pas beaucoup de sens (*the, is, and, of ...*),
- Ligne 8 : on applique la lemmatisation à chaque mot restant, c'est à dire qu'on ramène chaque mot à sa forme canonique.

Maintenant que le nettoyage des données a été effectué, nous passons à la vectorisation des mots. Nous verrons qu'il existe deux types de vectorisation.

2.2 La Vectorisation

La vectorisation est une étape cruciale en machine learning pour transformer du texte, en données numériques, exploitables par les algorithmes. En effet, les modèles de machine learning ne comprennent pas le texte, ils ont besoin de chiffres.

2.2.1 Vectorisation BAG-OF-WORD

Le principe de cette méthode est le suivant : le modèle *Bag of Words* (ou sac de mots) repose sur une idée simple ; on ignore l'ordre des mots dans un texte, et on garde que leur fréquence d'apparition.

Exemple concret :

Supposons qu'on ait 3 phrases :
- "Le chat dort"
- "Le chien dort"
- "Le chat mange"

D'abord, nous construisons le vocabulaire, c'est à dire qu'on crée une liste de mots uniques. Dans notre cas cela donnerait : ["Le", "chat", "dort", "chien","mange"].

Ensuite, on crée les vecteurs. On peut le représenter dans un tableau :

Mots	"Le chat dort"	"Le chien dort"	"Le chat mange"
Le	1	1	1
chat	1	0	1
dort	1	1	0
chien	0	1	0
mange	0	0	1

Ce qui est établi dans ce tableau, est le raisonnement suivant :

Si l'un des mots qui est compris dans la liste de vocabulaire, apparait dans le tweet étudié alors on lui attribut 1 point, s'il apparait 2 fois alors on lui attribut 2 points... Ainsi, nous pouvons apercevoir les mots qui reviennent le plus.

En supprimant les stopwords, comme nous l'avons fait au préalable pour nos tweets, nous pouvons déduire les mots qui sont les plus fréquents et qui sont les plus pertinents. Cependant, ce n'est pas garantie. En effet, même après avoir supprimé les stopwords, certains mots peuvent encore être génériques et fréquents, donc la fréquence seule ne

suffit pas à juger de la pertinence du mot. Nous nous rendons compte que la méthode BoW ne permet pas de mesurer précisément "l'importance des mots".

Néanmoins il existe d'autre technique de vectorisation qui peuvent donner des indices utiles sur les mots qui caractérisent un texte, c'est le cas de la méthode *TF-IDF*.

Code

```
bow_vectorizer = CountVectorizer(max_features=2500)
X_bow = bow_vectorizer.fit_transform(df_tweets['clean_text'])

#dataframe conversion

df_bow = pd.DataFrame(X_bow.toarray(),
columns=bow_vectorizer.get_feature_names_out())

#add label column
df_bow['label'] = df_tweets['label'].values
```

2.2.2 Vectorisation TF-IDF

TF-IDF (Term Frequency - Inverse Document Frequency), est une autre méthode de vectorisation, qui cherche aussi à connaitre l'importance des mots dans un tweet. L'idée de cette méthode est qu'un mot est important s'il est fréquent dans un tweet donné, mais peu fréquent dans un autre tweet.

Décomposition de cette méthode :

D'abord nous avons **TF** (Term Frequency), qui correspond à la fréquence du mot dans un document. Nous pouvons l'illuster par la formule suivante :

$$\mathrm{TF}(w, d) = \frac{\text{nombre d'occurrences du mot } w \text{ dans le document } d}{\text{nombre total de mots dans le document } d}$$

Puis nous avons IDF (Inverse Document Frequency), qui mesure la rareté d'un mot dans tous le corpus.

Que l'on peut illuster par la formule suivante :

$$\mathrm{IDF}(w) = \log\left(\frac{N}{1 + \mathrm{DF}(w)}\right)$$

où ;

$$N = \text{nombre total de documents}$$

$$DF(w) = \text{nombre de document contenant le mot } w$$

Le *+1* évite la division par zéro

Exemple concret :

Nous avons les trois "tweets" suivants :

1. "le chat dort"
2. "le chien dort"
3. "le chat mange"

Prenons l'exemple du mot "chat" :

- calcul de TF :

$$\text{TF}(w, d) = \frac{\text{nombre d'occurrences du mot } chat \text{ dans le tweet } 1.}{\text{nombre total de mots dans le tweet } 1.}$$

$$\text{TF}(\text{"}chat\text{"}, doc1) = \frac{1}{3} = 0.33$$

- calcul de DF :

$$\text{DF}(\text{"}chat\text{"}) = 2 (\text{car il apparaît dans 1. et 3.})$$

- calcul d'IDF :

$$\text{IDF}(\text{"}chat\text{"}) = log(\frac{3}{1+2}) = log(1) = 0$$

- calcul de TF-IDF :

$$\text{TF-IDF} = 0.33 * 0 = 0$$

Conclusion le mot n'est pas pertinent, c'est à dire qu'il ne permet pas de distinguer un tweet d'un autre. Cela signifie que "chat" apparait dans au moins 80 % des tweets, il n'est donc pas pertinent pour la classification, car trop générique dans le contexte du corpus(= colonne *text* dans notre contexte). Nous comprenons dorénavant que statistiquement le mot "chat" ne permet pas de distinguer un tweet d'un autre.

Code

```
        #Vectorization by TF-IDF

        tfidf_vectorizer = TfidfVectorizer(max_features=2500)
        X_tfidf = tfidf_vectorizer.fit_transform(df_tweets['clean_text'])

```

```
6         #dataframe conversion
7         df_tfidf = pd.DataFrame(X_tfidf.toarray(),
8         columns=tfidf_vectorizer.get_feature_names_out())
9
10        # best words
11        moyenne = df_tfidf.mean().sort_values(ascending=False)
12        print(moyenne.head(10))
13
14        #add label column
15        df_tfidf['label'] = df_tweets['label'].values
```

Explication du code pour la vectorisation TF-IDF :

— Lignes 1 et 2 : nous créons un objet `TfidfVectorizer` qui définit un « moule » capable de contenir jusqu'à 2500 mots. Nous limitons le vocabulaire avec `max_features=2500` afin de ne conserver que les mots les plus importants et éviter le bruit. Puis, nous assignons à `X_tfidf` la vectorisation TF-IDF du texte via la méthode `.fit_transform()`, où :

  — `fit` : apprend les mots du corpus (i.e. tous les mots présents dans les tweets),

  — `transform` : génère un vecteur TF-IDF pour chaque tweet (c'est-à-dire qu'il calcul les scores TF-IDF).

  `X_tfidf` devient donc une sorte de matrice dans laquelle nous avons deux colonnes *Coords* et *Values* qui se présentent comme suit :

```
Coords    Values
(0, 1162) 0.43942665471220743
(0, 1776) 0.5201519206547832
```

  $(0, 1162)$ dans *Coords* signifie que le tweet n°0 contient le mot n°1162 (numérotation du vocabulaire appris par TfidfVectorizer). Et 0,439426...contenu dans la colonne *Values* qui est le score TF-IDF associé au mot n°1162, score qui est au demeurant assez élevé, indique que ce mot est fréquent dans ce tweet mais rare dans les autres tweets.

— Lignes 7 et 8 : nous permettent de convertir la matrice `X_tfidf` en un dataframe pour une meilleure visualisation globale, avec pour chaque colonne, un mot (avec `.get_feature_names_out()` qui permet la conversion de la numérotation des mots en leur nom originel), associé à un tweet (numéroté), et relié par un score tf-idf :

	loser	loss	lost	lot	loui	love	low	lower	lowest	lp	ltd	luckin	lululemon
9511	0.32901	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
9512	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
9513	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.49278	0.00000
9514	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.43784

— Ligne 11 et 12 (optionnelles mais intéressantes) : nous faisons un classement des mots les plus importants en moyenne. Cela consiste à calculer la moyenne des scores TF-IDF pour chaque mot à travers tous les tweets à l'aide de la fonction suivante : `.mean().sort_values()`. Ainsi, nous avons le résultat suivant pour les 10 mots les plus importants :

```
stock       0.021490
report      0.015860
result      0.014932
market      0.012166
price       0.011788
us          0.011786
earn        0.011430
dividend    0.011210
buy         0.010870
new         0.010842
```

— Ligne 15 : nous ajoutons la colonne *label* dans notre dataset, qui nous informe sur le sentiment du tweet.

Résumé de la vectorisation :

Nous avons donc pu nous apercevoir qu'un score TF-IDF élevé indique qu'un mot est spécifique à un tweet, donc utile pour différencier ce tweet d'autre tweet. Mais TF-IDF ne donne pas le sens ou le sentiment de ce mot (même chose pour BoW), il nous renseigne sur le fait que le mot est important ici, car il est rare ailleurs.

Par exemple, pour la phrase suivante : "ce produit est exceptionnel, je suis très satisfait !" ; "exceptionnel" et "satisfait" sont rares et spécifiques, en plus ils sont positifs, mais ni TF-IDF ni BoW ne le savent.

Donc pour conclure cette première partie, la vectorisation est utile pour analyser les mots d'un tweet et les classifier, mais cela ne va pas nous permettre de dire si le mot est positif, neutre ou négatif. Pour ce faire, il nous faut un modèle de **Machine Learning** pour apprendre à faire le lien entre mot et sentiment.

3 Modèles

Après avoir préparé une base de donné, et appliquer les méthodes de vectorisation, nous pouvons maintenant passer à la partie **Machine Learning**. Nous verrons différents modèles de Machine Learning, en expliquant leurs mécanismes, pour ensuite pouvoir les comparer et choisir celui qui paraît être le meilleur. Ces modèles seront dans un premier temps binaires, c'est à dire qu'ils devront choisir si les tweets ont un sentiment négatif ou non négatif. Enfin, nous analyserons dans une dernière partie, un modèle multiclasse qui devra choisir si le tweet parait positif, neutre ou négatif.

3.1 Binaires

Puisque nous commençons par les modèles binaires, nous devons modifier la variable *label* qui contient trois élements différents. Nous allons donc regrouper les tweets qui ont un sentiments neutre (=1) et les tweets qui ont un sentiment positif (=2). Ainsi, la variable *label* sera codé de la manière suivante : 0 = négatif et 1 = non-négatif.

3.1.1 Logistic Regression

Mécanisme :
La régression logistique est un algorithme supervisé de classification binaire, utilisé ici pour prédire si un tweet appartient à la classe 1 (non-négatif, pertinent, etc.) ou à la classe 0 (négatif, non pertinent, etc.).

Une fois les tweets vectorisés (via TF-IDF ou Bag of Words), chaque tweet est représenté sous la forme d'un vecteur :

$$\mathbf{x} = (x_1, x_2, \ldots, x_n)$$

où chaque x_i représente le poids (fréquence ou importance) du mot i dans le tweet.
Le modèle calcule un score linéaire :

$$z = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^\top \mathbf{x} + w_0$$

et applique la fonction sigmoïde pour obtenir une probabilité :

$$\hat{y} = \frac{1}{1 + e^{-z}}$$

— Si $\hat{y} > 0.5$, le tweet est classé dans la classe 1.
— Sinon, il est classé dans la classe 0.

Les poids w_i sont appris automatiquement durant l'entraînement du modèle pour minimiser l'erreur de prédiction.

L'interprétation des poids est la suivante :

— Un **poids positif élevé** signifie que le mot est associé à la classe 1.
— Un **poids négatif** est associé à la classe 0.
— Un **poids proche de zéro** indique un mot peu informatif.

Exemple illustratif :

— Vecteur du tweet : $\mathbf{x} = [0,\ 0.42,\ 0,\ 0.88,\ 0,\ \ldots]$
— Poids appris : $\mathbf{w} = [0,\ 1.3,\ 0,\ -0.9,\ 0,\ \ldots]$

Calcul du score linéaire :

$$z = (0.42 \times 1.3) + (0.88 \times -0.9) = 0.546 - 0.792 = -0.246$$

Application de la fonction sigmoïde :

$$\hat{y} = \frac{1}{1 + e^{0.246}} \approx 0.44$$

Conclusion : le tweet est classé dans la classe 0 (car $\hat{y} < 0.5$).

Comment les poids sont appris ?

Lors de `model.fit[X_train, y_train]` (c.f : code) :

1. Le modèle essaie des poids initiaux (souvent aléatoires).
2. Il calcul la prédiction (via la fonction sigmoïde).
3. Il compare la prédiction au vrai label.
4. Il ajuste les poids pour minimiser l'erreur (avec une méthode comme la descente de gradient).

Ce processus est répété jusqu'à ce que le modèle converge (trouve des poids stables qui donnent de bonnes prédictions)

Qu'est-ce-que la descente de gradient ?

Dans notre projet, nous utilisons la régression logistique pour prédire si un tweet appartient à la classe 1 (pertinent, non-négatif , etc.) ou à la classe 0 (pertinent, négatif). Le modèle apprend les poids associés à chaque mot vectorisé (TF-IDF ou BoW) grâce à un algorithme d'optimisation : la **descente de gradient**.

Fonction de coût à minimiser :

La régression logistique ne cherche pas à minimiser une erreur quadratique, mais une fonction appelée *log-loss* ou entropie croisée :

$$J(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \quad \text{avec} \quad \hat{y}_i = \frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}_i}}$$

Principe de la descente de gradient :
Le modèle commence avec des poids aléatoires, puis les ajuste progressivement en suivant le sens opposé du gradient de la fonction de coût :

$$w_j^2 = w_j^1 - \alpha \cdot \frac{\partial J}{\partial w_j}$$

où :

- α est le **taux d'apprentissage** (learning rate),
- $\frac{\partial J}{\partial w_j}$ est la dérivée partielle de la fonction de coût.

Ce processus est répété jusqu'à convergence, c'est-à-dire jusqu'à ce que la fonction de coût devienne minimale.

Exemple concret dans notre contexte :

Soit un tweet vectorisé :

$$\mathbf{x} = [0,\ 0.42,\ 0,\ 0.88,\ 0,\ \ldots]$$

et les poids actuels :

$$\mathbf{w} = [0,\ 1.3,\ 0,\ -0.9,\ 0,\ \ldots]$$

Le score linéaire est :

$$z = 0.42 \times 1.3 + 0.88 \times (-0.9) = 0.546 - 0.792 = -0.246$$

Application de la fonction sigmoïde :

$$\hat{y} = \frac{1}{1+e^{0.246}} \approx 0.44$$

Le modèle prédit donc la **classe 0**. S'il s'était trompé (si la vraie étiquette était 1), la descente de gradient ajusterait les poids pour réduire cette erreur lors des prochaines itérations. En pratique avec `scikit-learn` la commande `model.fit(X_train, y_train)` appelle automatiquement un solveur (ex. `liblinear`, `lbfgs`) qui utilise la descente de gradient (ou une variante) pour entraîner le modèle.

Comment la descente de gradient ajuste les poids si la prédiction est mauvaise ?

Lorsqu'un tweet est mal classé par le modèle de régression logistique, la descente de gradient corrige les poids associés aux mots responsables de cette erreur.

Principe :
Soit un tweet vectorisé $\mathbf{x} = (x_1, x_2, ..., x_n)$, et une prédiction donnée par :

$$\hat{y} = \frac{1}{1 + e^{-z}} \quad \text{avec} \quad z = \mathbf{w}^\top \mathbf{x}$$

Si la prédiction est incorrecte (par exemple $\hat{y} = 0.2$ alors que $y = 1$), alors l'erreur $y - \hat{y}$ est grande et positive.

Chaque poids w_j est mis à jour comme suit :

$$w_j = w_j + \alpha \cdot (y - \hat{y}) \cdot x_j$$

où :

— α est le **taux d'apprentissage**,
— $(y - \hat{y})$ est le **sens de l'erreur**,
— x_j est l'importance du mot j dans ce tweet.

Exemple :

— Le mot "urgence" a une valeur $x_j = 0.9$ dans le tweet.
— Le modèle prédit $\hat{y} = 0.2$, alors que la vraie étiquette est $y = 1$.
— Donc : $y - \hat{y} = 0.8$

La mise à jour devient :

$$w_{\text{urgence}} = w_{\text{urgence}} + \alpha \cdot 0.8 \cdot 0.9$$

Cela signifie que le poids du mot "urgence" augmente : il deviendra plus influent pour prédire la classe 1 à l'avenir.

Cas inverse :
Si le modèle prédit $\hat{y} = 0.9$ alors que $y = 0$, les poids associés aux mots présents seront **diminués** pour éviter de refaire la même erreur.

Code

```
        import pandas as pd
        df = pd.read_csv("cleaning/tweets_bow.csv")

        import time
        from sklearn.linear_model import LogisticRegression
        from sklearn.model_selection import train_test_split
```

```
        from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, f1_score, precision_score, reca


        df['label'] = df['label'].apply(lambda x: 0 if x == 0 else 1)

        X = df.drop("label", axis=1)
        y = df["label"]

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123)


        start_time = time.time()
        model = LogisticRegression(solver='liblinear', random_state=42)

        model.fit(X_train, y_train)

        y_pred = model.predict(X_test)
        end_time = time.time()
        elapsed_time = end_time - start_time
```

Explication du code : Régression Logistique

- Lignes 1 à 2 : chargement du fichier `tweets_bow.csv` contenant les tweets vectorisés (par la méthode Bag of Words). Chaque ligne représente un tweet, chaque colonne un mot (feature), et la colonne `label` contient l'étiquette de classification.
- Ligne 10 : on remplace tous les labels différents de 0 par 1, afin de faire une classification binaire (0 = classe négative, 1 = classe non négative).
- Lignes 12 à 13 :
  — `X` contient les variables explicatives (les mots, donc toutes les colonnes sauf `label`),
  — `y` contient la variable cible (la colonne `label`).
- Ligne 15 : on divise les données en deux groupes :
  — 70% pour l'entraînement (`X_train`, `y_train`),
  — 30% pour le test (`X_test`, `y_test`).
- Lignes 19 à 21 :
  — on initialise un modèle de régression logistique (`LogisticRegression`) avec le solveur `liblinear`, adapté aux petites matrices creuses (comme avec Bag of Words),
  — `.fit()` permet d'entraîner le modèle sur les données d'entraînement.
- Ligne 23 : prédiction du label des tweets de test.
- Lignes 18 et 24 : mesure du temps total d'exécution (entraînement + prédiction).

Résultats :

Table 1 – Classification report du modèle de régression logistique

Classe	Précision	Rappel	F1-score	Support
Classe 0	0.7535	0.3830	0.5078	423
Classe 1	0.9014	0.9783	0.9383	2440
Accuracy			**0.8903**	2863
Macro avg	0.8275	0.6806	**0.7231**	2863
Weighted avg	0.8796	0.8903	**0.8747**	2863

3.1.2 K-Nearest Neighbors

Mécanisme :

1. **Principe général** :

   K-NN ne construit pas de modèle à proprement parler. Pour prédire la classe d'un tweet inconnu, l'algorithme :

   (a) Calcule la **distance** entre ce tweet et tous les tweets du jeu d'entraînement (ex : distance euclidienne).

   (b) Sélectionne les K tweets les plus proches (les K-plus proches voisins). C'est un choix arbitraire, mais attention au bruit et au risque d'erreur.

   (c) Attribue la classe majoritaire parmi ces voisins à l'exemple à prédire.

2. **Fonctionnement dans notre projet** :

   Chaque tweet est représenté comme un vecteur $\mathbf{x} \in \mathbb{R}^n$, avec n mots.

   Soit un tweet inconnu $\mathbf{x}_{\text{test}}$, et une base d'entraînement contenant des tweets $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m$, chacun avec un label $y_i \in \{0, 1\}$.

   Le modèle calcule :

   $$\text{dist}(\mathbf{x}_{\text{test}}, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{n} (x_{\text{test},j} - x_{i,j})^2}$$

   puis choisit les K plus petites distances et vote à la majorité.

   Exemple de calcul de distance :

3. **Exemple** :

   Objectif : Prédire le sentiment d'un tweet à l'aide de l'algorithme KNN ($K = 3$), après vectorisation Bag of Words (BoW).

   Données d'entraînement :

   — Tweet A : "great product fast delivery" Label : 1 (positif)
   — Tweet B : "terrible service never again" Label : 0 (négatif)
   — Tweet C : "fast service and great experience" Label : 1 (positif)

   Tweet à prédire : "great fast service"

   Construction du vocabulaire :

   On extrait tous les mots uniques :

   Vocabulaire = ["great", "product", "fast", "delivery", "terrible", "service", "never", "again", "and", "experience"]

   Vectorisation BoW :

Chaque tweet est transformé en vecteur binaire (1 si le mot est présent, 0 sinon) :

$$\text{Tweet A} = [1, 1, 1, 1, 0, 0, 0, 0, 0, 0]$$
$$\text{Tweet B} = [0, 0, 0, 0, 1, 1, 1, 1, 0, 0]$$
$$\text{Tweet C} = [1, 0, 1, 0, 0, 1, 0, 0, 1, 1]$$
$$\text{Tweet test} = [1, 0, 1, 0, 0, 1, 0, 0, 0, 0]$$

Calcul des distances euclidiennes :

- Distance avec Tweet A : $\sqrt{(1-1)^2 + (0-1)^2 + (1-1)^2 + (0-1)^2} = \sqrt{2} \approx 1.41$
- Distance avec Tweet B : $\sqrt{(1-0)^2 + (1-1)^2 + (0-1)^2 + (0-1)^2 + \dots} = \sqrt{5} \approx 2.24$
- Distance avec Tweet C : $\sqrt{(0-1)^2 + (0-1)^2} = \sqrt{2} \approx 1.41$

Classification (K = 3) :

On prend les 3 voisins les plus proches :

- Tweet A (distance 1.41, label 1)
- Tweet C (distance 1.41, label 1)
- Tweet B (distance 2.24, label 0)

On essaie toujours de choisir des K impairs pour éviter les égalités.

$\Rightarrow$ 2 votes positifs, 1 vote négatif $\Rightarrow$ **Prédiction finale** : **1 (positif)**

4. **Particularités du modèle** :
   - **Pas de phase d'entraînement** : toute la "mémoire" du modèle réside dans les données d'entraînement.
   - **Très sensible aux échelles et à la densité locale** des points.
   - **Coût élevé en prédiction** : on doit recalculer les distances à chaque nouvelle instance.

5. **Choix du paramètre K :**
   - K trop petit : modèle sensible au bruit (overfitting),
   - K trop grand : risque d'erreur (underfitting),
   - En pratique, on choisit K via validation croisée.

Avantage dans notre projet :

K-NN permet de classer un tweet en se basant uniquement sur la proximité avec d'autres tweets vectorisés, sans nécessiter d'apprentissage complexe. Il est simple, interprétable, mais plus lent à l'usage.

Code

```python
import pandas as pd
import numpy as np
#%%
df_bow = pd.read_csv(r'C:\Users\nolev\Desktop\M1 Econometrics\Machine Learning\Data\tweets_bow.csv')
#%% md
Binary classification
#%%
df_bow['binary_label'] = df_bow['label'].apply(lambda x: 0 if x == 0 else 1) # 0 if negagative 1 else
#%% md
Splitting the data for train/test
#%%
X = df_bow.drop(columns=['label', 'binary_label'])
y = df_bow['binary_label']
#%%
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
#%% md
Implement KNN
#%%
#Libs
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
import time
#%%
start_time = time.time()

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(x_train, y_train)
y_pred = knn.predict(x_test)

end_time = time.time()
elapsed_time = end_time - start_time
```

Résultats :

Table 2 – Classification report du modèle K-Nearest Neighbors

Classe	Précision	Rappel	F1-score	Support
Classe 0	0.5775	0.2865	0.3830	377
Classe 1	0.8777	0.9607	0.9173	2009
Accuracy			**0.8541**	2386
Macro avg	0.7276	0.6236	**0.6501**	2386
Weighted avg	0.8302	0.8541	**0.8329**	2386

Temps d'entrainement : < 1 seconde

Explication du code de l'algorithme K-Nearest Neighbors (KNN)

1. **Chargement des données vectorisées (BoW)**

   On utilise un fichier CSV contenant les tweets déjà vectorisés par Bag of Words :

   ```
   df_bow = pd.read_csv('tweets_bow.csv')
   ```

2. **Transformation en classification binaire**

   La variable cible initiale peut contenir plusieurs niveaux. On la convertit en une variable binaire :

   ```
   df_bow['binary_label'] = df_bow['label'].apply(lambda x: 0 if x
   == 0 else 1)
   ```

   Cela permet de distinguer les tweets **négatifs (0)** des autres (**positifs ou neutres** $\rightarrow$ **1**).

3. **Séparation des données**

   On isole les variables explicatives X et la cible y, puis on divise le jeu de données en ensemble d'entraînement (75%) et de test (25%) :

   ```
   X = df_bow.drop(columns=['label', 'binary_label'])
   y = df_bow['binary_label']
   x_train, x_test, y_train, y_test = train_test_split(X, y,
   test_size=0.25, random_state=42)
   ```

4. **Entraînement et prédiction avec KNN**

   On utilise le classificateur `KNeighborsClassifier` de `scikit-learn` avec $K = 1$:

   ```
   knn = KNeighborsClassifier(n_neighbors=1)
   knn.fit(x_train, y_train)
   y_pred = knn.predict(x_test)
   ```

   Le modèle est **non paramétrique** : il stocke les exemples d'entraînement et classe chaque observation test en fonction des K voisins les plus proches.

3.1.3 Random Forest

Mécanisme :

1. **Présentation du modèle**

   Le modèle **Random Forest** est un ensemble (*ensemble learning*) de plusieurs **arbres de décision**, chacun construit aléatoirement sur des sous-échantillons du jeu de données. La prédiction finale est obtenue par un **vote majoritaire** des arbres.

2. **Fonctionnement dans le cadre du projet**

   — Chaque tweet est transformé en vecteur via la méthode BoW : chaque dimension représente la présence (ou fréquence) d'un mot du vocabulaire.
   — Un grand nombre d'arbres est généré. Pour chaque arbre :
      — Un **échantillon aléatoire** des tweets est sélectionné avec remise (bootstrap).
      — À chaque nœud de l'arbre, une **sélection aléatoire de mots** (variables) est testée pour trouver la meilleure séparation.
      — L'arbre est construit jusqu'à certains critères d'arrêt (ex. : profondeur maximale, nombre minimal d'échantillons par feuille).
   — Lors de la prédiction, chaque arbre vote pour une classe (0 ou 1) pour un tweet donné. Le résultat final est la **classe majoritaire parmi tous les arbres**.

3. **Intérêts du modèle Random Forest pour ce projet**

   — **Robuste** aux mots non pertinents : tous les mots ne sont pas utilisés à chaque arbre.
   — **Réduction du surapprentissage** par rapport à un arbre de décision seul.
   — **Capable d'identifier les mots importants** pour la prédiction (analyse des *feature importantes*).
   — **Non paramétrique** : aucune hypothèse forte sur la distribution des données.

4. **Exemple d'interprétation (importance des mots)**

   Contrairement à un modèle à base de règles logiques, Random Forest ne cherche pas à détecter des mots directement "non-négatifs" ou "négatifs". Il s'appuie sur des **associations statistiques** entre la présence de certains mots et les sentiments observés dans les données d'entraînement.

   Par exemple, le mot `"tesla"` peut paraître neutre. Mais le modèle peut apprendre que :

   — Si un tweet contient `"tesla"` **et** `"plunges"` $\Rightarrow$ souvent **négatif**.
   — Si un tweet contient `"tesla"` sans mots alarmants, il est souvent **positif**.

Ainsi, même un mot neutre peut être utile s'il permet de structurer une règle combinée pertinente. L'arbre peut alors apprendre la logique suivante :

$$\text{Si “tesla” présent} : \begin{cases} \text{et “plunges” présent} \Rightarrow \text{négatif} \\ \text{sinon} \Rightarrow \text{positif} \end{cases}$$

En combinant des milliers de telles règles partielles dans la forêt, le modèle devient capable de capturer des motifs complexes et efficaces pour la classification.

Code

```
tweet_bow = pd.read_csv(r"C:\Users\tweets_bow (1).csv")

tweet_bow['label_binary'] = tweet_bow['label'].apply(lambda x: 0 if x==0 else 1)
tweet_bow = tweet_bow.drop(columns=['label'])

from sklearn.model_selection import train_test_split

X = tweet_bow.drop(columns=['label_binary'])
y = tweet_bow['label_binary']

X_train, X_test, y_train, y_test = train_test_split(
X, y,
test_size=0.2,
stratify=y,
random_state=42
)

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

rf_tuned = RandomForestClassifier(
bootstrap=True,
random_state=42,
n_estimators=200,
class_weight='balanced',
max_depth=None,
min_samples_leaf=1,
min_samples_split=5
)

import time
start = time.time()
rf_model.fit(X_train, y_train)
end = time.time()

y_pred = rf_model.predict(X_test)

print('Classification Report:\n', classification_report(y_test, y_pred, digits=4))
print("Temps d'entraînement :", round(end - start, 2), "secondes")...
```

Explication du modèle Random Forest pour la classification de sentiments de tweets

1. **Séparation du jeu de données** :

   On sépare les variables explicatives X et la variable cible y, puis on effectue une division **stratifiée** en jeu d'entraînement (80%) et jeu de test (20%) :

   ```
   X_train, X_test, y_train, y_test = train_test_split(
   X, y,
   test_size=0.2,
   stratify=y,
   random_state=42
   )
   ```

   La stratification permet de conserver la même proportion de classes dans les deux sous-échantillons.

2. **Entraînement du modèle Random Forest** :

   On instancie un classificateur `RandomForestClassifier` avec 100 arbres :

   ```
   rf_model = RandomForestClassifier(
   n_estimators=100,
   max_depth=None,
   random_state=42
   )
   ```

3. **Prédiction et évaluation** :

   On applique le modèle sur le jeu de test, puis on génère un rapport de classification :

   ```
   y_pred = rf_model.predict(X_test)
   print(classification_report(y_test, y_pred, digits=4))
   ```

Résultats :

Table 3 – Classification report du modèle Random Forest

Classe	**Précision**	**Rappel**	**F1-score**	**Support**
Classe 0	0.7081	0.4549	0.5539	288
Classe 1	0.9089	0.9667	0.9369	1621
Accuracy			**0.8895**	1909
Macro avg	0.8085	0.7108	**0.7454**	1909
Weighted avg	0.8786	0.8895	**0.8791**	1909

Temps d'entrainement : 12.92 secondes

3.1.4 Multilayer Perceptron

Mécanisme :

Le **Multilayer Perceptron**, souvent appelé modèle réseau fonctionne à travers 3 couches différentes. Plus précisément, une couche d'entrée (Vectorisation du Tweet), des petites couches cachées (couches intermédiaires avec des fonctions d'activations ; fonction ReLu...), et enfin une couche de sortie (fonction sigmoïde qui donne le résultat du sentiment).

Petit exemple :

Tweet vectorisé : [0,1,1,0,...,0] (couche 1)
Ce tweet vectorisé va passer dans les couches intermédiaires (détail ci-après) (couche 2), pour finir dans la couche de sortie, pour nous donner le sentiment du tweet :

sigmoïde = 0,84 , alors sentiment positif car 0,84>0,5

Exemple concret :
Contexte :
Nous avons le vocabulaire suivant : ["tesla", "plunges","profits"]
Notre tweet étudié est le suivant :

"tesla profits"

Vectorisé cela donne : x=[1,0,1] (car "tesla" et "profits" sont présents, "plunges" n'est pas présent dans le vocabulaire)
Nous avons notre **couche 1**, la couche d'entrée.

Supposons maintenant, une couche caché avec 2 neurones et une fonction d'activation $ReLU$: Nous avons 3 entrées (vectorisation BoW) x=[1,0,1] ; avec, nous avons dit, 2 neurones. Cela donne une matrice 3:2 avec 6 poids :

$$W^{(1)} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}$$

Avec cette matrice nous comprenons les choses suivantes :

- w_{11} : poids entre l'entrée 1 et le neurone caché 1
- w_{32} : poids entre l'entrée 3 et le neurone caché 2
- etc..

Dans cet exemple nous allons donner des poids arbitrairement, bien que dans la réalité l'alogrithme ajuste les poids, nous verrons plus tard comment.

Chaque neurone contient un ensemble de poids w_i (un par entrée) et un biais (valeur ajoutée après la somme pondérée des entrées). Le biais permet au neurone d'ajuster son seuil d'activiation même si toutes les entrées sont nulles.

Nous avons les composantes suivantes :

$$W^{(1)} = \begin{pmatrix} 0{,}5 & -1{,}0 \\ -0{,}4 & 0{,}3 \\ 0{,}8 & 0{,}6 \end{pmatrix} \quad \text{et} \quad b^{(1)} \begin{pmatrix} 0{,}1 & 0{,}2 \end{pmatrix}$$

Nous pouvons maintenant calculer les activations cachées (**couche 2** : fonction ReLU) :

$$h = ReLU \times (W^{(1)} \times \mathtt{x} + b^{(1)})$$

- $$W^{(1)} \times \mathtt{x} = \begin{pmatrix} 0{,}5.1 + -0, 4.0 + 0, 8.1 \\ -1{,}0.1 + 0{,}3.0 + 0{,}6.1 \end{pmatrix} = \begin{pmatrix} 0{,}5 + 0, 8 \\ -1 + 0, 6 \end{pmatrix} = \begin{pmatrix} 1{,}3 \\ -0{,}4 \end{pmatrix}$$
- Ajout des biais :

$$\begin{pmatrix} 1{,}3 \\ -0{,}4 \end{pmatrix} + \begin{pmatrix} 0{,}1; 0{,}2 \end{pmatrix} = \begin{pmatrix} 1{,}3 + 0{,}1 \\ -0{,}4 + 0, 2 \end{pmatrix} = \begin{pmatrix} 1{,}4 \\ -0{,}2 \end{pmatrix}$$

- Nous pouvons maintenant utiliser la fonction $ReLU$ (**couche 2**) :

$$h = \begin{pmatrix} ReLU(1, 4) \\ ReLU(-0, 2) \end{pmatrix} = \begin{pmatrix} 1{,}4 \\ 0 \end{pmatrix}$$

Explication fonction ReLU :
La fonction ReLU est une activation couramment utilisée dans les réseaux de neurones. elle est définit mathématiquement comme suit :

$$f(\mathtt{x}) = max(0, \mathtt{x})$$

Cela signifie que si l'entrée $\boldsymbol{x}$ *est négative, la sortie sera (0) et si* $\boldsymbol{x}$ *est positive ou nulle, la sortie sera* $\boldsymbol{x}$

Enfin, nous pouvons calculer le sentiment du tweet (**couche 3**) :
Couche de sortie :

$$Poids : W^{(2)} = \begin{pmatrix} 1{,}2 \\ -0{,}7 \end{pmatrix} ; Biais : b^{(2)} = 0{,}1$$

Calcul des activations cachées :

$$\mathrm{z} = \sum_{j=1}^{m} (w_{(j)}^{(2)} \times h_{(j)} + b^{(2)}) = \sum_{j=1}^{m} \begin{pmatrix} 1{,}2 \\ -0{,}7 \end{pmatrix} \times \begin{pmatrix} 1{,}4 \\ 0 \end{pmatrix} + 0{,}1 = 1{,}2 \times 1{,}4 + (-0{,}7) \times 0 + 0{,}1 = 1{,}78$$

Cela nous permet de réaliser la fonction sigmoïde :

$$\hat{y} = \frac{1}{1 + e^{(-1{,}78)}} = 0,855$$

Résultat : Le réseau prédit un score de 0,855 le tweet est donc classé positif car, 0,855 $>$ 0,5.

Compréhension du système de poids dans le réseau de neurone :

Il existe trois grandes étapes :

1. **Initialisation aléatoire** : Au départ, tous les poids W et biais b sont initialisés aléatoirement (avec de petites valeurs proches de 0). Le modèle commence sans a priori : il ne sait rien.
2. **Propagation avant (forward pass)** :
   — Calcul de la fonction sigmoïde.
   — Comparaison de la sortie prédite avec la vraie valeur du tweet (0 ou 1).
3. **Rétropropagation et descente de gradient** : C'est lors de cette étape que les poids sont ajustés. Elle se décompose en trois sous-parties :
   — **Calcul de la fonction de perte** :

$$\text{Loss} = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

   — **Calcul du gradient** : Dérivées partielles de la perte par rapport à chaque poids et biais.
   — **Mise à jour des poids** : Les poids sont ajustés dans la direction qui réduit l'erreur (descente de gradient).

Exemple concret d'ajustement des poids :
Objectif : Montrer comment les poids et biais sont ajustés pendant une itération d'apprentissage surpervisé à partir de :

— un tweet vectorisé
— un label (0 ou 1)
— une fonction de perte
— Une retropropagation simplifiée

Etape 1 : Initialisation Nous mettons en place des poids et biais initiaux aléatoirement/arbitrairement :

- Poids d'entrée vers couche caché (un poids par mot) :

$$W^{(1)} = [0{,}5; 0{,}2; -0{,}3]$$

- Biais de la couche cachée :

$$b^{(1)} = 0{,}1$$

- Poids caché :

$$W^{(2)} = 0{,}4$$

- Biais de sortie :

$$b^{(2)} = 0{,}2$$

Etape 2 : Propagation avant

- Calcul sortie couche cachée :

$$\begin{aligned} h &= \sum x_i \cdot w_i^{(1)} + b^{(1)} \\ &= (1 \cdot 0{,}5) + (0 \cdot 0{,}2) + (1 \cdot -0{,}3) \\ &= 0{,}3 \end{aligned}$$

- Calcul de la sortie finale (sigmoïde) :

$$\begin{aligned} z &= h \cdot W^{(2)} + b^{(2)} \\ &= 0{,}3 \cdot 0{,}4 + 0{,}2 \end{aligned}$$

$$\begin{aligned} \hat{y} &= \sigma(0{,}32) \\ &= \frac{1}{1 + e^{(-0{,}32)}} \\ &= \mathbf{0{,}579} \end{aligned}$$

Le modèle prédit un sentiment plutôt positif, car $\hat{y} = 0{,}579$ (car $\hat{y} > 0{,}5$).

Etape 3 : Rétropropagation et ajustement :

- **Calcul de l'erreur (fonction de perte : binaire croisée)**

$$Loss = -[ylog(\hat{y}) + (1-y)log(1-\hat{y})]$$

On sait que y est égal à 1 !

$$\begin{aligned} Loss &= -[1.log(\hat{y}) + (1-1)log(1-\hat{y})] \\ &= -[log(\hat{y})] \\ &= -log(0{,}579) \\ &= 0,547 \end{aligned}$$

- **Calcul du gradient de la perte par rapport à z.**

$$\frac{\partial Loss}{\partial z} = \hat{y} - y = 0{,}579 - 1 = -0{,}421$$

Démonstration :

On cherche à montrer l'égalité suivante :

$$\boldsymbol{\frac{\partial \text{Loss}}{\partial z^{(2)}} = \frac{\partial \hat{y}}{\partial z} \times \frac{\partial \text{Loss}}{\partial \hat{y}} = \hat{y} - y}$$

1. *Calcul de*

$$\frac{\partial \hat{y}}{\partial z} \quad \text{avec} \quad \hat{y} = \frac{1}{1+e^{(-z)}} = (1+e^{(-z)})^{-1}$$

<=>

$$\begin{aligned} \frac{\partial \hat{y}}{\partial z} &= -1(1+e^{(-z)})^{-1-1} \cdot \frac{d}{dz}(1+e^{-z}) \\ &= -(1+e^{-z})^{-2} \cdot (-e^{-z}) \\ &= \frac{1}{-(1+e^{z})^2} \cdot -e^{-z} \\ &= \frac{-e^{-z}}{-(1+e^{-z})^2} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \end{aligned}$$

<=>

Nous avions $\hat{y} = \frac{1}{1+e^{-z}}$ et $1 - \hat{y} = \frac{e^{-z}}{(1+e^{-z})}$.

<=>

Nous pouvons en déduire :

$$\hat{y}(1-\hat{y}) = \frac{1}{(1+e^{-z})^2} = \frac{\partial \hat{y}}{\partial z}$$

2. *Calcul de*

$$\frac{\partial Loss}{\partial \hat{y}} \quad \text{avec} \quad Loss = -[ylog(\hat{y}) + (1-y)log(1-\hat{y})]$$

<=>

$$\begin{aligned}\frac{\partial Loss}{\partial \hat{y}} &= -[y\frac{1}{\hat{y}} + (1-y)\frac{-1}{(1-\hat{y})}] \\ &= -[\frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})}]\end{aligned}$$

3. *Calcul de*

$$\frac{\partial Loss}{\partial z^{(2)}} = \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial Loss}{\partial \hat{y}}$$

<=>

$$\begin{aligned}\frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial Loss}{\partial \hat{y}} &= \hat{y}(1-\hat{y}) \cdot [-(\frac{y}{\hat{y}} - \frac{(1-y)}{(1-\hat{y})})] \\ &= \hat{y}(1-\hat{y}) \cdot [-(\frac{y(1-\hat{y}) + (1-y)\hat{y})}{\hat{y}(1-\hat{y})})] \\ &= (1-y)\hat{y} - y(1-\hat{y}) \\ &= \boldsymbol{\hat{y} - y}\end{aligned}$$

- **Mise à jour des poids**

  Poids cachés :

  $\triangle W^{(2)} = \eta \cdot \frac{\partial Loss}{\partial z} \cdot h = 0{,}1 \cdot (-0{,}421) \cdot 0{,}3 = -0{,}0126$
  $W^{(2)}_{New} = 0,4 - 0,0126 = \mathbf{0,3874}$
  $\triangle b^{(2)} = 0{,}1 \cdot (-0{,}421) = \eta \cdot \frac{\partial Loss}{\partial z} = -0{,}0421$
  $b^{(2)}_{New} = 0,2 - 0,0421 = 0,1579$

  Poids d'entrée :

  Mot "***tesla***" :

  $\triangle W^{(1)}_1 = 0{,}1 \cdot (-0,421) \cdot 0{,}4 \cdot 1 = -0,0168$

  $W^{(1)}_1 = 0{,}5 - 0{,}0168 = 0{,}4832$

  Mot "***profits***" :

  $\triangle W^{(1)}_3 = 0{,}1 \cdot (-0,421) \cdot 0{,}4 \cdot 1 = -0,0168$

  $W^{(1)}_3 = -0{,}3 - 0{,}0168 = -0{,}3168$

Tableau de résultat du changement des poids et biais après itération :

Table 4 – Résutats Avant/Après itération

Poids/Biais	**Ancien**	**Nouveau**
$W_1^{(1)}$	0.5	0.4832
$W_3^{(1)}$	-0.3	-0.3168
$W^{(2)}$	0.4	0.3874
$b^{(2)}$	0.2	0.1579

-> Ce processus se répète pour tous les tweets du datasets, pendant plusieurs itérations.

Questions :

1. Quand est-ce que le processus doit-il s'arrêter ?
2. Pourquoi les poids varient si peu ?
3. Si les poids aléatoires (étape 1) donnent déjà une bonne prédiction, est-ce qu'on doit quand même rétropropager ?

1. Il existe plusieurs stratégies :
   - Nombre d'époque/itérations fixé :

     ```
     model.fit(X_train,y_train, epochs = 20)
     ```

   - Convergence de la perte : On surveille la fonction de perte (Loss) -> si la perte ne diminue plus suffisamment, on considère que le modèle a fini d'apprendre.
   - Early stopping (arrêt anticipé) : On s'arrête, si les performances sur un jeu de validation cessent de s'améliorer. Par exemple, le réseau s'entraîne pendant 100 époques maximum, mais si la val_loss (perte sur les données de validation) n'a pas diminué pendant 5 époques, on interrompt l'entraînement. Cela permet d'éviter le surapprentissage (overfitting).
2. Il existe plusieurs raisons :
   - Stabilité de l'apprentissage : Le learning rate ($\eta = 0.1$ dans l'exemple) est conçu pour que les poids évoluent petit à petit. On ne veut surtout pas de grands sauts, car cela rendrait l'apprentissage instable, voire divergent.
   - Contrôle de l'erreur : dans l'exemple on a $\hat{y} = 0.579$ pour $y = 1$, donc l'erreur $\hat{y} - y = -0.456$ est modérée, on fait un petit ajustement des poids pour pousser $\hat{y}$ un peu plus vers 1. Ce petit pas corrige l'erreur sans surcorriger.
   - Lissage du modèle : Le réseau est entrainé sur beaucoup d'exemples. Chaque exemple fait une petite mise à jour. C'est l'accumulation de ces petits ajuste-

ments, sur plusieurs époques, qui permet au réseau de converger vers des poids optimaux.

3. N'a t'on plus besoin de rétropropager, si les poids de l'étape 1, prédisent déjà la bonne classe de sentiment ?

   — L'objectif n'est pas juste "avoir la bonne classe", mais aussi de réduire la perte ! Même si la classe est correcte, la fonction de perte : $Loss = -log(0.89) = 0.116$. Elle n'est pas nulle, donc le modèle peut encore s'améliorer en rendant sa prédiction plus proche de 1. Par exemple, une prédiction de 0.999 serait plus confiant.

   — La rétropropagation sert à affiner tous les poids. Même si le tweet actuel est bien classé, d'autres exemple dans le batch ne le sont peut-être pas ? On met à jour les poids pour améliorer la performance globale sur l'ensemble des tweets.

Code :

```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
import time

start_time = time.time()

mlp_tuned = MLPClassifier(random_state=42, hidden_layer_sizes=(100,), max_iter=200, activation= 'relu', solver='lbfg
mlp_tuned.fit(x_train, y_train)
y_pred_tuned = mlp_tuned.predict(x_test)

end_time = time.time()
elapsed_time = end_time - start_time
```

Résultats :

Table 5 – Classification report du modèle MLP

Classe	Précision	Rappel	F1-score	Support
Classe 0	0.5994	0.5199	0.5568	377
Classe 1	0.9121	0.9348	0.9233	2009
Accuracy			**0.8692**	2386
Macro avg	0.7557	0.7273	**0.7401**	2386
Weighted avg	0.8627	0.8692	**0.8654**	2386

Temps d'entrainement : 56 secondes

3.1.5 Support-Vector Machine

Mécanisme :

- Qu'est ce qu'un *SVM* ?
  Un algorithme de classification supervisée qui cherche à trouver la meilleur frontière (ou hyperplan) pour séparer les données entre deux classes (ex : non-négatif VS négatif).

- Principe de base : trouver le bon hyperplan :
  Le *SVM* cherche une frontière qui sépare les classes (ex : tweets non-négatifs vs négatifs), et maximise la marge, c'est à dire la distance entre la frontière et les points les plus proches de chaque classe (=vecteurs de support).

- La frontière :

  $$w.\mathrm{x} + b = 0$$

  tel que :

  $$y_i(w \cdot \mathrm{x}_i + b) >= 1$$

  tout en minimisant :

  $$1/2||w||^2 = 1/2(w_1^2 + w_2^2 + w_3^2)$$

- Utiliser ce modèle est un avantage, puisqu'il est efficace en haute dimension (parfait avec BoW ou TF-IDF), et il généralise bien (grâce à la marge maximale). En revanche il est plus lent sur des grands jeux de données et moins performant si les données qui sont très "bruitées".

Exemple :

Vocabulaire BoW = ["tesla", "profits","plunges"]

Table 6 – Données d'entrainement

Tweets	**Vector**	**Sentiment**
"tesla profits"	[1,1,0]	1
"tesla plunges"	[1,0,1]	0

Nous allons entrainer un SVM liénaire pour trouver les poids $W = [w_1, w_2, w_3]$ et un biais b tels que :

$$f(X) = W \cdot X + b$$
$$\text{sous la contrainte que :}$$
$$y_i(W \cdot x_i + b) >= 1$$

Application :

- Tweet 1 : "tesla profits", $f(x)_1 = [1, 1, 0], y_1 = 1$

$$<=> 1 \cdot (w_1 + w_2 + b) >= 1$$
$$<=> w_1 + w_2 + b >= 1 \text{ (Eq 1)}$$

- Tweet 2 : "tesla plunges", $f(x)_2 = [1, 0, 1], y_2 = 0$ => SVM reformule le label avec $y = (-1; +1)$ donc $y_2 = -1$

$$<=> -1 \cdot (w_1 + w_3 + b) >= 1$$
$$<=> -(w_1 + w_3 + b) >= 1 \text{ (Eq 2)}$$

Donc grâce à ces deux équations et la contrainte qui est la suivante :

$$\frac{1}{2}||w||^2 = \frac{1}{2}(w_1^2 + w_2^2 + w_3^3)$$

Nous allons pouvoir trouver les poids et biais, en résolvant la problème d'optimisation avec par exemple en Lagrangien :

$$L(W, b, \alpha) = \frac{1}{2}||W||^2 - \sum_{i=1}^{n} \alpha_i[y_i(W \cdot X_i + b) - 1]$$

(on se rappelle que le code fait toutes ces étapes automatiquement !)

Les solutions trouvées sont :

$$W = [1; 1{,}5; -2] \text{ et } b = -1$$

Vérification dans les contraintes :

— Tweet 1 : $1 \cdot ([1 + 1{,}5] + (-1)) = 1{,}5 >= 1$
— Tweet 2 : $-1([1 + (-2)] + (-1)]) = -1 \cdot (-2) = 2 >= 1$

Maintenant, nous pouvons tester un tweet inconnu :

Tweet : "profits plunges"

$$f(x)_{test} = [0, 1, 1]$$

Calcul du score :

$$\begin{aligned} f(x) &= 1 \cdot 0 + 1{,}5 \cdot 1 + (-2) \cdot 1 + (-1) \\ &= 1{,}5 - 2 - 1 \\ &= -1{,}5 \end{aligned}$$

Comme $f(x) < 0$; le SVM prévoit que le tweet soit négatif (classe 0).

Code

```
from sklearn.svm import SVC

svm_model = SVC(
kernel='linear',
probability=True,
random_state=42
)

start = time.time()
svm_model.fit(X_train, y_train)
end = time.time()

y_pred_svm = svm_model.predict(X_test)

print('Classification Report:\n',classification_report(y_test, y_pred_svm, digits=4))
print("Temps d'entraînement :", round(end - start, 2), "secondes")
```

Résultats :

Table 7 – Classification report du modèle SVM

Classe	Précision	Rappel	F1-score	Support
Classe 0	0.8471	0.3404	0.4857	423
Classe 1	0.8964	0.9893	0.9406	2440
Accuracy			**0.8771**	2863
Macro avg	0.7590	0.7158	**0.7342**	2863
Weighted avg	0.8682	0.8771	**0.8715**	2863

Temps d'entrainement : 151 secondes

3.1.6 FinBert

Vectorisation BertTokenizer :
Pour le bon fonctionnement de l'algorithme, FinBERT on a besoin d'une vectorisation que l'on appelle BertTokenizer.

En quoi consiste-elle ?

a) **Fonctionnement concret du `BertTokenizer`**
Prenons comme exemple le tweet suivant :

`"Tesla shares fall after disappointing earnings"`

- **Objectif de la tokenisation** : Transformer ce texte en vecteurs d'entrée exploitables par le modèle FinBERT. Cela comprend plusieurs étapes :
  — Segmentation en sous-mots (WordPieces)
  — Ajout des tokens spéciaux : `[CLS]` (début) et `[SEP]` (fin)
  — Encodage en IDs numériques
  — Création du masque d'attention
- **Code Python** :

```python
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("Yiyanghkust/finbert-tone")

tweet = "tesla shares fall after disappointing earnings"
tokens = tokenizer(tweet, padding="max_length", truncation=True,
max_length=20, return_tensors="pt")
```

- **Tokenisation textuelle (WordPieces)** :
  Tokenisation "classique" :

```python
tokenizer.tokenize(tweet)
```

`['tesla', 'shares', 'fall', 'after', 'disappointing', 'earnings']`

  En réalité, BERT découpe en WordPieces :

`['tesla', 'shares', 'fall', 'after', 'dis', '##app', '##oint', '##ing', 'earnings']`

- **Explication des WordPieces** :
  (a) `##app` indique qu'il s'agit d'un fragment rattaché au mot précédent. Le symbole `##` signifie que ce n'est pas un mot autonome.
  (b) La tokenisation en WordPieces permet à FinBERT de traiter des néologismes ou termes inconnus, fréquents dans le domaine financier.
  (c) Le modèle est robuste face à des mots nouveaux ou composés, ex. `crypto-liquidity` devient `['crypto', '-', 'liqu', '##idity']`

(d) Cette stratégie permet de garder un vocabulaire limité, évitant un token unique pour chaque mot du langage.

- **Ajout des tokens spéciaux** :

  => [CLS] tesla shares fall after dis ##app ##oint ##ing earnings [SEP]

  [CLS] : sert de représentation globale de la séquence. Dans les tâches de classification (comme l'analyse de sentiment, la détéction de spam...), le vecteur de sortie correspondant à ce token est utilisé comme entrée pour le classifieur final.

  [SEP] : Positionné en fin de phrase ou entre deux phrases si on fait une tâche bi-phrase (tâches NLI : *Natural language Inference*). Cela permet d'indiquer au modèle où une phrase se termine et où l'autre commence.

- **Encodage numérique** : Chaque token est transformé en un entier (input_ids) correspondant à son index dans le vocabulaire FinBert : Par exemple pour mon tweet :
  => [101, 12345, 4567, 7890, 8765, 234, 567, 890, 4321, 101]

- **Masque d'attention** : Les modèle comme BERT attendent des longueurs de phrase (=tensors) de taille fixe (ex : max_length = 20 tokens). par exemple si un tweet fait 12 tokens, on lui ajoute 8 zéros pour que sa longueur atteigne 20.
  Nous avons donc un `inputs_ids` :

  [101, 2456, 3241, 1187, 7423, 102, 0,0, ...,0]

  Et un `attention_mask` :

  [1,1,1,1,1,1,0,0,...,0]

  -> le modèle ignore les 0 !

  Ce qui vient d'être fait s'appelle le ***Padding***, il permet de traiter tous les tweets ensemble sans que leurs longueurs posent problème. S'assure que le modèle ne s'appuie que sur les vrais mots pour apprendre ou prédire.

  Ca y est nous avons tous nos tweets vectorisés ! !

b) **Encodage du tweet par FinBert** :

- **Objectif** :
  Prendre le tweet vectorisé par BerTokenizer (avec ses `input_ids` et `attention_mask`) et produire une représentation profonde et contextuelle du sens du tweet, que l'on pourra utiliser pour prédire son sentiment.

- **Encodage contextuel par FinBert** :

  Embedding : Chaque "input_id" (token) est transformé en un vecteur dense de taille 768.

  Que contient ce vecteur dense ?

  Chaque vecteur encode plusieurs types d'information :
  — sémantique du mot : ex : "fall" $\neq$ "rise"
  — Position dans la phrase (via l'embedding positionnel)
  — Type de segment (utile pour les paires de phrase)

  Exemple fictif : On prend "*Tesla*" comme Token :
  Vecteur dense du token Tesla :

  [0,14,-0,33,0,08, ..., 0,72] <- 768 valeurs au total

  => Ce n'est pas interprétable humainement mais le modèle apprend que ce vecteur est "proche" de ceux de "Apple", "Nvidia" ou "Elon Musk" dans le sens sémantique.

  Encodeur Transformer :

  Ensuite chaque token passera dans des couches BERT, 12 au total dans notre cas. Chaque couche est un encodeur *Transformer*.

  Une couche BERT est composé de 4 sous-couches :

  — **Mutli-Head Self Attention** : Pour que chaque token "regarde" les autres et s'attende lui-même plusieurs fois sans différents angles.
  — **Add & Norm** : Opération de normalisation résiduelle : **Add** : on ajoute l'entrée originale (résidual correction) ; **Norm** : on applique une *Layer Normalisation*, cela aide à stabiliser l'entrainement et à faciliter la propagation du gradient.
  — **Feed-Forwoard Network (MLP)** : une MLP positionnelle appliquée à chaque token individuellement.
  — **Add & Norm** (encore) : Même principe, on ajoute l'entrée précédente et on normalise à nouveau.

  => Ces 4 "sous-couches" jouent un rôle crucial dans le traitement du texte et l'apprentissage de représentation contextuelles profondes.

  Exemple :
  Contexte :

Phrase d'entrée :

"The bank collapsed yesterday"

Tokenization du tweet : ['[CLS]', 'the', 'bank', 'collapsed', 'yesterday','.','[SEP]']

Nous avons ensuite nos `input_ids` puis `attention_mask`. Puis, chaque mot/-token est transformé en vecteur dense :

Par exemple bank :

[0,12, -0,45, ..., 0,78] # vecteur de taille 768

C'est donc à partir de ce moment là, que les tokens passent dans les couches (12), qui sont composées de 4 sous-couches :

- **Multi-Head Self Attention** :

  Le mot "*bank*" va regarder tous les autres mots de la phrase, y compris lui-même. Le modèle calcul une pondération d'attention vers chaque token :

  Table 8 – Pondération d'attention vers chaque token

Token	**Pondération (score d'attention)**
[CLS]	0.05
"the"	0.10
"bank"	0.20
"collapsed"	0.50
"yesterday	0.10
"."	0.05

  On remarque que "collapsed" a un score plus élevé que les autres mots ! Pour quelle raison ? Parce que ce mot aide le modèle à comprendre le sens de "bank" (...institution financière). Donc ici, l'attention contextualise "bank" en captant son lien avec "collapsed". Le vecteur de "bank" est mis à jour par une moyenne pondérée des vecteurs de tous les autres tokens.

- **Add & LayerNorm** :

  Le nouveau vecteur de "bank" est ajouté au vecteur d'origine (résidual connection). Puis, on applique une normalisation (LayerNorm) pour stabiliser les valzurs et accélérer l'apprentissage. Résultats : un vecteur toujours de taille 768, mais avec de nouvelles valeurs enrichies par le contexte.

- **Feed-Forward Network (MLP)**

  On passe les tokens dans un MLP, comme vu dans l'explication du modèle Multilayer-Perceptron. C'est une transformation non-linéaire indépendante des autres tokens, mais qui complexifie la représentation.

- **Add & LayerNorm** :

  On réapplique un résidual : ajout l'entrée du MLP à sa sortie, puis on applique une normalisation.

=> Nous avons donc maintenant un nouveau vecteur final pour "bank", qui contient maintenant :

— signification du mot "bank" selon son contexte local
— une représentation enrichie grâce au MLP
— Une stabilisation via le *LayerNorm*.

- **Résumé** :
  Au début : "*bank*" -> vecteur appris au hasard (embedding statistique). Après 1 couche BERT : "*bank*" -> vecteur contextuel, influencé par "*collapsed*.
  Si on passe "*bank*" dans 12 couches, chaque couche renforce et affine cette compréhension, jusqu'à ce qu'on ait une représentation ultra contextualisée.

c) **Entrainement supervisé de FinBERT** :

- **Objectif** : Ajuster les poids du modèle, notamment la couche de classification finale, à partir de tweets labellisés pour que FinBERT approuve à reconnaître le sentiment associé à chaque tweet.

- **Code** :

```
from transformers import Training Arguments, Trainer

training_args = TrainingArguments(
output_dir="./finbert_output",
evaluation_strategy="epoch",
save_strategy="epoch",
num_train_epochs=2,
per_device_train_batch_size=16,
per_device_eval_batch_size=16,
learning_rate=2e-5,
warmup_steps=100,
weight_decay=0.00,
logging_dir="./logs",
logging_steps=10,
load_best_model_at_end=True,
seed=42
)
```

Que fait cette configuration ?

— Entrainement pendant 2 époques (2 passages sur le tout le jeu d'entrainement)
— 16 tweets vus à la fois (`batch_size`)
— Optimisation via Adam W (standard pour BERT)

— Taux d'apprentissage de 2e-5 (petit pour ne pas trop modifier les poids pré-entrainés)
— sauvegarde automatique du meilleur modèle (load_best_model_at_end=True).

- **Code de lancement d'entrainement** :

```
trainer = Trainer(
model=model,
args= training_args,
train_dataset=dataset_split["train"],
eval_dataset=dataset_split["test"]
)
```

Que se passe-t-il ici ?

(a) Chaque tweet du train set est : tokenisé, encodé par FinBERT, et passé à travers la couche de classification

(b) Le modèle compare la prédiction avec le label réel (via la *CrossEntrepy-Loss*).

(c) Le modèle ajuste ses poids pour que la prochaine prédiction soit meilleure. Cela modifie la couche linéaire finale, et légèrement les couches internes de BERT.

d) **Éclaircissement du lien entre b) et c)** :

L'objectif de c) est d'ajuster les poids internes de FinBERT (surtout dans la couche de classification) pour que ses prédictions se rapprochent des vrais labels.

Pour ce faire :

1. Comparaison de la prédiction avec la vérité :

   Contexte :

   Notre tweet :

   "Tesla shares fall after disappointing earnings"

   Son label est 0, donc son sentiment est négatif. Ensuite, FinBERT va produire des logits, qui sont des valeurs de sortie de la dernière couche linéaire du modèle avant d'appliquer une fonction comme *softmax* :

   Exemple concret de calcul des logits :

   Vecteur résumé : $h_{CLS} = [0.3, -0.4, 0.6, 0.1]$ (tweet encodé)

   Poids et biais de la couche de sortie :

$$W = \begin{pmatrix} 0{,}2 & -0{,}5 & 0{,}7 & -0{,}3 \\ -0{,}4 & 0{,}6 & -0{,}6 & 0{,}2 \end{pmatrix} \quad \text{et} \quad b^{(1)} \begin{pmatrix} 0{,}1 & -0{,}2 \end{pmatrix}$$

   Calcul des logits :

$$logits = W \cdot h_{CLS} + b$$

Classe 0 :

$$\begin{aligned} logits_0 &= 0{,}2(0{,}3) + (-0{,}5)(-0{,}4) + 0{,}7(0{,}6) + (-0{,}3)(0{,}1) + 0{,}1 \\ &= 0{,}06 + 0{,}20 + 0{,}42 - 0{,}03 + 0{,}1 \\ &= 0{,}75 \end{aligned}$$

Classe 1 :

$$\begin{aligned} logits_1 &= (-0,4)(0{,}3) + 0{,}6(-0{,}4) + (-0{,}6)(0{,}6) + 0{,}2(0{,}1) - 0{,}2 \\ &= -0{,}12 - 0{,}24 - 0{,}36 + 0{,}02 - 0{,}2 \\ &= -0{,}90 \end{aligned}$$

Résultat :

$logits = [0{,}75; -0{,}90]$

Transformation en probabilité via *softmax* :

$$\begin{aligned} \hat{y}_0 &= \frac{e^{0{,}75}}{e^{0{,}75} + e^{-0{,}90}} \\ &\approx \frac{2{,}12}{2{,}12 + 0{,}41} \\ &\approx 0{,}838 \end{aligned}$$

$\hat{y}_1 = 1 - \hat{y}_0 = 0{,}162$

=> Bonne prédiction ! !

2. Comparaison au label réel :

   Le vrai label est 0 (négatif), donc on veut que $\hat{y}_0$ soit proche de 1.

   Fonction de perte :

   $L = -log(\hat{y}_{true}) = -log(0{,}838) \approx 0{,}176$

   Plus la prédiction est fausse, plus la perte est grande ! ! !

3. Retropropagation :

   Ajustement des poids pour, que le score de la bonne classe augmente, et que, celui de la mauvaise classe diminue.

   Nous avons démontré précèdemment comment l'ajustement des poids pouvait être effectué (voir plus haut dans les différentes démonstration).

4. Répète sur tout le dataset :

   Code :

```
trainer = Trainer(
model = model,
```

```
                        args=training_args,
                        train_dataset =  dataset_split["train"],
                        eval_dataset = dataset_split["test"]
                        )
                trainer.train()
```

Le modèle passe sur tout le jeu d'entrainement (2 fois) et ajuste ses poids pour mieux prédire les sentiments.

e) **Evaluation Finale** :

Le but ici est de tester le modèle sur les tweets de test (jamais vus pendant l'entrainement) pour voir à quel point il généralise bien.

Code pour l'évaluation :

```
from sklearn.metrics import classification_report

predictions = trainer.predict(dataset_split["test"])

y_true = predictions.label_ids
y_pred = np.argmax(predictions.predictions, axis=1)

print(classification_report(y_true, y_pred, target_names=['Negative', 'Non-negative'], digits=4))
```

Ce qu'il se passe en coulisse :

(a) `trainer.predict(...)` :

- FinBERT encode chaque tweet du jeu test
- génère un vecteur [CLS], passe dans la couche linéaire
- Produit des logits, convertis en probabilités

(b) `np.argmax(...)` : Le modèle choisit la classe avec la plus forte probabilité

(c) Comparaison avec les vrais lables (`label_ids`) : on évalue la précision, rappel, F1-score.

Rappel :

1. Precision : parmi les tweets que le modèle a prédit négatifs, combien l'étaient vraiment
2. Rappel : parmi les vraies tweets négatifs, combien le modèle en a trouvés ?
3. F1-score : moyenne entre précision et rappel
4. Accuracy : % global de tweets bien classés

Différence entre c) et e) :

c) - Entrainement : ici, le modèle voit les tweets d'entrainement avec leurs labels. Il apprend via la fonction de perte, la retropropagation, ect..

e) - Évaluation finale : ici le modèle ne reçoit que les tweets, il prédit les labels. Ensuite, on compare ses prédictions aux vraies labels avec `classification_report`

Résultats :

Table 9 – Classification report du modèle FinBERT

Classe	Précision	Rappel	F1-score	Support
Classe 0	0.7122	0.5317	0.6088	363
Classe 1	0.9196	0.9614	0.9401	2023
Accuracy			**0.8961**	2386
Macro avg	0.8159	0.7466	**0.7745**	2386
Weighted avg	0.8881	0.8961	**0.8897**	2386

Temps d'entrainement : 247 secondes

3.1.7 Tableau comparatifs des résultats de l'ensemble des modèles binaires

Table 10 – Comparaison des résultats des modèles binaires

Modele	Précision	Rappel	F1-score	Support
Weighted avg_logit	0.8796	0.8903	0.8747	2863
Weighted avg_KNN	0.8302	0.8541	0.8329	2386
Weighted avg_RF	0.8786	0.8895	**0.8791**	1909
Weighted avg_MLP	0.8627	0.8692	0.8654	2386
Weighted avg_SVM	0.8682	0.8771	0.8715	2863
Weighted avg_FinBERT	0.8881	0.8961	**0.8897**	2386

- On remarque que les modèles binaires qui ont le meilleurs F1-score sont le ***Random Forest*** et le ***FinBERT***, suivi de près par le modèle ***Logits*** étonnement car c'est le moins sophistiqué.
- Est-ce que le modèle multiclasse aura un meilleur score ? Nous pouvons penser intuitivement que les scores seront moins élevés puisqu'il y a plus de classe à prédire (neutre, positif, négatif). Nous verrons cela dans la partie suivante.

3.2 Multiclasses

3.2.1 Expansion des données

L'objectif est de doubler le nombre de tweets disponibles pour l'entraînement, en générant des reformulations anglaises à partir des tweets originaux, sans changer leur étiquette de sentiment.

Étapes détaillées :

1. **Chargement des tweets**
   Les tweets originaux sont extraits d'un fichier Excel. Les URLs sont supprimées et les labels sont remappés dans l'intervalle `[0, 1, 2]` pour une classification multi-classe.
2. **Initialisation des modèles de traduction**
   Deux modèles MarianMT préentraînés (de la bibliothèque `Helsinki-NLP`) sont chargés :
   — `opus-mt-en-fr` pour traduire les tweets de l'anglais vers le français.
   — `opus-mt-fr-en` pour effectuer la traduction retour.
3. **Traduction par lots**
   Les tweets sont traités en petits lots (batches). Chaque batch est :
   — Tokenisé puis traduit vers le français.
   — Puis traduit de nouveau vers l'anglais.

   Ce processus donne une version anglaise *reformulée* de chaque tweet, tout en conservant son sens.
4. **Back-translation (traduction aller-retour)**
   Chaque tweet original est enrichi par une version légèrement différente (formulation alternative), générée automatiquement par passage en français puis retour en anglais.
5. **Construction du dataset final**
   Pour chaque tweet :
   — Le tweet original est conservé.
   — Sa version back-traduite est ajoutée.
   — Les deux lignes partagent le même label.

   Le dataset final contient donc le **double** de tweets, permettant un meilleur apprentissage du modèle de classification.

Exemple :
Tweet original : *"Tesla shares fall after earnings"*
Tweet back-traduit : *"Tesla stocks drop following poor results"*
Label : `0` (négatif)

3.2.2 ROBERTA MODELE MULTICLASS

Rappel du pré-traitement des données :
Etape d'augmentation des données via la traduction automatique + back-translation :

Le but de cette va petre de générer des nouvelles versions de tweets enrichies sémantiquement (sans charger leur sens), pour doubler le dataset et améliorer la robustesse d'un futur modèle de classification multiclasse.

Pour ce faire, nous allons utiliser deux modèles `MarianMT`, qui permettent de faire de la traduction automatique anglais -> français, français -> anglais, dans le cadre d'une augmentation de données par `Back-translation`.

Exemple visuel

Tweet d'origine :

"Tesla shares drop sharly after earnings miss"

Le modèle `en-fr` pourrait traduire :

"Les actions Tesla chutent fortement après des résultats décevants"

Puis le modèle `fr-en` redonne une version légérement différente du tweet d'origine :

"Tesla stock falls significantly after disappointing results"

On voit que le sens est conservé, mais les mots changent !!!

Nous utilisons cette méthode, car elle permet d'augmenter les données de manière robuste, ainsi le back-translation génère des exemples réalistes et variés de tweets, sans modifier leur signification.

Modèle RoBERTa Multiclasse :

Etape 1 : Lecture du dataset et prétraitement :

- Chargement fichier CSV
- Supression des URLs
- Remappage des labels

Cela garanti que les tweets sont propres, uniformisés, et que les labels sont adaptés pour une classification multiclasse avec les outils de *Hugging Face*.

Etape 2 : Calcul automatique des poids de classes :

Objectif : Ajuster l'entrainement pour que le modèle ne privilégie pas les classes majoritaires

Pourquoi faire ?

Lorsqu'un jeu de données est déséquilibré, un modèle peut apprendre à prédire principalement la classe la plus fréquente (biais de majorité), en ignorant les classes minoritaires. Pour corriger ce biais, on utilise des poids que l'on assigne à chaque classe.

Explication :

1. Comptage des tweets par classe : On compte le nombre d'exemple pour chaque label (0, 1, 2). Résultat potentiel : 0 (négatif) = 2 000 ; 1 (neutre) = 5 000 ; 2 (positif) = 3 000

2. Calcul global :
   `total_samples = len(df)` -> nombre total de lignes (tweets) `num_classes = len(label_counts)` -> nombre de classes (= 3 ici)

3. Formule de pondération :

$$\text{computed\_weights} = \frac{\text{total\_samples}}{\text{num\_classes} \cdot \text{label\_counts}}$$

$$W_0 = \frac{N}{K \cdot n_0} = \frac{10\,000}{3 \cdot 2\,000} = 1.6666$$
$$W_1 = \frac{N}{K \cdot n_1} = \frac{10\,000}{3 \cdot 5\,000} = 0.6666$$
$$W_2 = \frac{N}{K \cdot n_2} = \frac{10\,000}{3 \cdot 3\,000} = 1.1111$$

   Cette formule donne plus de poids aux classes rares, et moins aux classes fréquentes.

4. Conversion en tenseur PyTorch
   Afin de corriger le déséquilibre entre les classes du jeu de données, on attribue à chaque classe un poids proportionnel à l'inverse de sa fréquence. Ces poids sont ensuite convertis en **tenseur PyTorch** pour être utilisés dans la fonction de perte `CrossEntropyLoss` :

$$\text{poids}_i = \frac{N_{\text{total}}}{C \cdot N_i}$$

   où N_{total} est le nombre total d'exemples, C le nombre de classes, et N_i le nombre d'exemples de la classe i.

Exemple : si les classes 0, 1 et 2 contiennent respectivement 400, 800 et 200 tweets, les poids calculés sont :

$$[1.17,\ 0.58,\ 2.33]$$

Ils sont ensuite convertis en tenseur :

```
class_weights = torch.tensor([1.17, 0.58, 2.33], dtype=torch.float)
```

Ce tenseur est utilisé lors de l'entraînement pour pénaliser davantage les erreurs sur les classes minoritaires :

```
loss_fct = torch.nn.CrossEntropyLoss(weight=class_weights)
```

Etape 3 : Chargement du modèle RoBERTa multiclasse :

Objectif :
Configurer et charger un modèle RoBERTa pré-entrainé pour qu'il soit, adapté à une classification multiclass, et, prêt à être entrainé sur des données de tweets financiers en anglais.

Code

```
model_checkpoint = "cardiffnlp/twitter-roberta-base-sentiment-latest"

config = RobertaConfig.from_pretrained(model_checkpoint)
config.num_labels = num_classes

tokenizer = RobertaTokenizer.from_pretrained(model_checkpoint)

model = RobertaForSequenceClassification.from_pretrained(
model_checkpoint,
config=config,
from_tf=True,
ignore_mismatched_sizes=True
)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Détail :

1. Ici, on séléctionne le modèle RoBERTa pré-entrainé :

```
model_chekpoint = "..."
```

C'est un modèle qui vient de Hugging Face, il est entrainé sur des tweets, et est une version affinée (fine-tuné) de RoBERTa adapté au langage de Twitter (hashtags, abréviations, ect...).

2. Chargement de la configuration du modèle

```
config = RobertaConfig.from_pretrained(model_checkpoint)
config.num_labels = num_classes
```

On modifie la configuration pour passer en mode classification multiclasse.

3. Chargement du tokenizer

```
tokenizer = RobertaTokenizer.from_pretrained(model_checkpoint)
```

Le tokenizer RoBERTa transforme les textes en vecteurs d'IDS de tokens compatibles avec ce modèle.

4. Chargement du modèle RoBERTa avec classification

```
model = RobertaForSequenceClassification.from_pretrained(
model_checkpoint,
config=config,
from_tf=True,
ignore_mismatched_sizes=True
)
```

5. Passage sur le bon device (GPU ou CPU) :

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Le modèle est déplacé en mémoire GPU si disponible (beaucoup plus rapide pour l'entrainement).

Etape 4 : Tokenization du dataset :

La logique générale de vectorisation est identique entre FinBERT et RoBERTa. Cependant, il existe une petite spécification entre ces deux modèles, en effet, les details du tokenizer et du vocabulaire changent, car ils dépendent du corpus d'entrainement (tweets vs textes financiers) et du modèle de base (BERT VS RoBERTa).

Code

```
def tokenize_function(examples):
texts = [str(text) for text in examples["text"]]
return tokenizer(texts, truncation=True, max_length=512)

dataset = Dataset.from_pandas(df)
tokenized_dataset = dataset.map(tokenize_function, batched=True)
```

Etape 5 : Trainer personnalisé avec *Cross Entropy Loss* pondérée :

Objectif :
Créer une version personnalisée de `Trainer` pour injecter les poids de classes dans la fonction de perte. Cette étape est indispensable pour gérer le déséquilibre entre classe (ex : si on a beaucoup plus de tweets neutres que positifs ou négatifs).

Détail :

— Classe personnalisée `WeightedLossTrainer`
Pourquoi personnaliser ?
Parce que `Trainer` standard n'applique pas de poids dans la fonction de perte par défaut, et nous voulons que les classes minoritaires soient plus fortement pénalisées en cas d'erreur.

— Méthode `compute_loss()`
C'est la fonction que est appelée à chaque étape d'entrainement :
  — Récupération des "labels" :
  — Passage dans le modèle
  — Définition de la fonction de perte avec poids
  — Calcul de la perte

— Retour du loss (+outputs si demandé)

```
return (loss, outputs) if return_outputs else less
```

Pourquoi cette étape est essentielle ? Sans cette pondération le modèle pourrait" jouer la sécurité" en prédisant toujours la classe dominante, l'accuracy augmenterait mais le F1-score serait faible sur les classes minoritaires.

Etape 6 : Définition des métriques d'évaluation :

Objectif :

Définir des métriques de performance personnalisées que seront calculées à chaque évaluation, et utilisées pour sélectionner le meilleur modèle pendant l'entrainement.

=> c'est une sorte de `Classification_report`

Etape 7 : Définition des métriques d'évaluation :

Objectif :

Dans un premier temps nous allons évaluer le modèle sur plusieurs splits de données, pour ensuite obtenir une estimation robuste des performances sur l'ensemble du dataset, enfin nous sauvegarderons les meilleurs modèles de chaque pli (fold).

Exemple concret :

Contexte fictif :

Imaginons que nous avons un dataset de 12 tweets multiclasse après augmentation (via back-translation) avec les classes suivantes :

ID	**Texte (résumé)**	**Label**
1	"Tesla beats earnings"	2 (positif)
2	"Apple stock falls sharply"	0 (négatif)
3	"Market remains stable despite Fed news"	1 (neutre)
4	"Tesla beats earnings" (version back-translatée)	2
5	"Apple stock falls sharply" (version BT)	0
6	"Market stable Fed news" (BT)	1
7	"Amazon revenue soars"	2
8	"Inflation fears hit Dow"	0
9	" Wall Street waits for Fed rate decision	1
10	"Amazon revenue soars" (BT)	2
11	"Inflation hits Dow" (BT)	0
12	"Fed rate announcement expected" (BT)	1

Etape de la validation croisée :

Stratified K-Fold Split :

- > `n_splits=3`, c'est le nombre de séparation du Dataset. Nous comprenons qu'ici nous séparons notre DataSet en 3, et chaque séparation contient le même proportion de labels [0, 1, 2].

SPLIT 1 :

- Apprentissage des labels par rapport aux tweets suivants (Train set) : [1, 2, 3, 4, 5, 6, 7, 8]
- Il se teste sur les 4 derniers (Validation set) : [9, 10, 11, 12]

Ainsi, le modèle s'entraîne sur les 8 premiers exemples avec la loss pondérée. Et à chaque époque il est évalué sur les 4 exemples de validation.
Cela donnerai la chose suivante :

Table 11 – Evaluation fold 1 / split 1

ID	True Label	Pred Label
9	1	1
10	2	2
11	0	1
12	1	1

Ce qui donnerait par exemple une `accuracy =75%, F1-score = 0.78`

SPLIT 2 :
Nous aurions donc un découpage différent des tweets d'apprentissage et des tweets de tests. Nous aurions alors de nouveaux scores. Ainsi, nous répéterions ce processus pour chaque découpage, dans notre cas nous avons pris 10 folds. Nous aurions alors 10 modèles fine-tunés, et nous pourrions choisir le modèle qui a les meilleurs scores.

Résumé de la Pipeline Roberta Multiclasse avec Back-Translation :

1. Préparation des données (back-translation)

   Objectif : enrichir le dataset avec de nouvelles versions paraphrasées.
   Méthode : Chaque tweet est traduit anglais -> français -> anglais via `MarianMT`. Pour chaque tweet original, on ajoute un tweet reformulé, mais avec le même label.
   Résultat : un dataset doublé, plus robuste aux variations d'expressions.

2. Pré traitement et équilibrage des classes

   Nettoyage : Suppression des liens (regex) , mapping des labels multiclasses (1->2, 2->1, 3->2, pour [0, 1, 2]).
   Calcul des poids de classes : $poids_i \frac{N_t otal}{C \cdot N_i}$ où N_i = nombre d'échantillon dans la classe i, et C = nombre de classes
   Objectif : compenser le déséquilibre dentre classes dans la loss

3. Chargement du modèle RoBERTa pré-entrainé :

   Modèle : `cardiffulp/twitter-roberta-base-sentiment-latest`
   Pourquoi ce modèle ? Il est pré entrainé sur des tweets, et est bien adapté au langage informel/financier.

Adaptation : on remplace la tête de classification par une couche à 3 neurones (pour 3 classes).

4. Tokenisation :

   Utilise le tokenizer RoBERTa : Transforme chaque tweet en `input_ids` + `attention_mask` , troncatureà 512 tokens max.

   Format : Hugging Face Dataset compatible avec Trainer.

5. Entrainement avec loss pondérée

   `CrossEntropyLoss(weight = class_weights)`

   Entrainement via une classe `WeightedLossTrainer` personnalisée.

   Avantage : Plus d'attention est portée aux classes rares, Réduction du biais vers la classe dominante (ex :"neutre")

6. Évaluation personnalisée :

   Métriques calculées à chaque époque : Accuracy, Precision, Recall, F1-Score pondéré

   Fonction `compute_metrics()` branchée dans le Trainer.

7. Validation croisée (`StratifiedKFold`) :

   10 splits du dataset avec stratification (même distribution des classes dans chaque pli).

   Pour chaque pli : Entrainement d'un modèle RoBERTa neuf, Evaluation sur les données de validation et sauvegarde des poids du meilleur modèle.

   Objectif : Evaluer la robustesse générale du modèle sur l'ensemble des données, et éviter l'overfitting sur un seul jeu de test.

8. Sauvegarde et rapport final :

   Modèles et tokenizers sauvegardés par pli, affichage des scores de chaque pli, calcul de la moyenne finale des performances.

Résultat :

Table 12 – Modèle RoBERTa fold 10

eval_loss	**eval_accuracy**	**eval_precision**	**eval_recall**	**eval_weighted_F1-score**
0,3394	0,9497	0,9498	0,9497	0,9497

Temps d'entrainement : 8h-10h

4 Conclusion : Quelle utilité nous avons à prédire le sentiment de tweet financier ?

L'analyse automatique du sentiment dans les tweets financiers présente de nombreuses applications concrètes dans le domaine économique et financier. Parmi les plus pertinentes, on peut citer :

— **Anticipation des mouvements de marché** : les tweets influents - en particulier ceux émis par entreprises, analystes ou personnalités - peuvent impacter rapidement le cours des actions, la volatilité d'un indice ou la confiance des investisseurs. Un modèle de prédiction de sentiment permet de détecter des signaux haussiers ou baissiers avant qu'ils ne se reflètent dans les prix.

— **Amélioration des stratégies de trading algorithmique** : le sentiment prédit peut être utilisé comme variable d'entrée dans des algorithmes de trading. Par exemple, un sentiment positif peut suggérer une position longue, tandis qu'un sentiment négatif peut inciter à la vente.

— **Surveillance de la réputation des entreprises** : les services marketing ou relations investisseurs utilisent l'analyse de sentiment pour, identifier une crise potentielle en ligne, réagir plus rapidement à des bad buzz, ou surveiller la perception publique avant/après une annonce financière.

— **Complément aux analyses traditionnelles** : l'analyse de sentiment enrichit les approches fondamentales ou techniques classiques, en ajoutant une dimension comportementale aux décisions d'investissement.

— **Utilisation pédagogique ou exploratoire** : les chercheurs peuvent étudier l'évolution du discours autour d'une entreprise ou analyser l'impact d'événements économiques majeurs à travers les émotions exprimées sur les réseaux sociaux.