



Université Sultan Moulay Slimane
École Nationale des Sciences Appliquées de Khouribga
Informatique et Ingénierie des Données (IID2)

PROJET TUTOIRE

*AI-Based Emotion and Stress Detection System
for Mental Well-Being Monitoring*

Réalisé par

CAMARA Famakan
AGOULZI Mohamed
EL ABBASSI Moad

Encadré par :

Pr. Meyem MAFTOUH

Année Universitaire : 2025/2026

Résumé

Ce rapport présente la conception et la réalisation d'un système de détection automatique des émotions et du stress à partir de données textuelles relatives à la santé mentale. Fondé sur une démarche visant à comparer plusieurs paradigmes d'Intelligence Artificielle, le système repose sur cinq approches de classification : la **Régression Logistique**, le **SVM linéaire**, le **BiLSTM**, **BERT** et **DistilBERT**.

Ces modèles sont entraînés et évalués dans des conditions expérimentales rigoureuses sur le dataset GOEMOTIONS (28 classes, 196,337 échantillons après augmentation) et le dataset HUMAN STRESS PREDICTION (augmenté à 8,000 échantillons équilibrés via WordNet). Une analyse comparative approfondie met en évidence les compromis entre précision, coût computationnel et latence d'inférence.

Les modèles sont intégrés dans une application web complète composée d'une API REST (Flask) et d'une interface utilisateur moderne (React,/,Vite), permettant la prédiction en temps réel, la visualisation des tendances par graphiques (Recharts) et la détection de motifs émotionnels à risque. Afin de respecter les limites du domaine médical, le système ne fournit aucun diagnostic médical et s'inscrit exclusivement dans un cadre éducatif et de recherche.

Mots-clés : traitement automatique du langage naturel (NLP), classification d'émotions, détection de stress, Transformers, BERT, BiLSTM, SVM, Flask, React, santé mentale.

Table des matières

Résumé	i
1 Introduction et Cahier des Charges	1
1.1 Contexte et motivation	1
1.2 Objectifs du projet	1
1.3 Organisation du rapport	1
2 Données et Prétraitement	3
2.1 Dataset 1 : GoEmotions (classification multi-classes)	3
2.2 Dataset 2 : Human Stress Prediction (classification binaire)	4
2.3 Augmentation de données	4
2.4 Découpage des jeux de données	5
2.5 Construction des labels single-label	5
2.6 Pipeline de nettoyage textuel	6
2.7 Stratégies de tokenisation	6
3 Modélisation et Architectures	8
3.1 Machine Learning classique (baselines)	8
3.1.1 Vectorisation TF-IDF	8
3.1.2 Régression Logistique	8
3.1.3 LinearSVC	9
3.2 Deep Learning : BiLSTM	9
3.3 Transformers : BERT et DistilBERT	11
3.3.1 BERT (110 M paramètres)	12
3.3.2 DistilBERT (66 M paramètres)	12
3.3.3 Fonction générique de fine-tuning	12
3.3.4 Lancement des deux fine-tunings	12
4 Résultats et Analyse Comparative	14
4.1 Protocole d'évaluation	14
4.2 Détection des 28 émotions (GoEmotions)	14
4.3 Détection du stress (dataset augmenté à 8 000 échantillons)	15
4.4 Complexité computationnelle	15
4.5 Recommandations par cas d'usage	16
5 Architecture Logicielle et Système	17
5.1 Architecture globale	17
5.2 Backend — API REST Flask	17

5.2.1	Point d'entrée et initialisation	17
5.2.2	Endpoints de l'API	18
5.2.3	Route de prédiction (exemple)	18
5.2.4	Classe EmotionPredictor (chargement multi-modèle)	19
5.2.5	Classe StressPredictor	19
5.3	Base de données SQLite	21
5.4	Frontend — Interface React	22
5.4.1	Technologies	22
5.4.2	Routage de l'application	22
5.4.3	Pages de l'application	23
6	Identification des Patterns de Risque	26
6.1	Scoring des patterns émotionnels	26
6.2	Algorithme de calcul du score de risque	26
6.3	Système d'alertes automatiques	27
6.4	Analyse temporelle du stress	28
6.5	Limites éthiques	29
	Conclusion et Perspectives	30
A	Structure du projet	31
B	Guide d'installation	33

Table des figures

5.1	Capture de la page Dashboard	23
5.2	Capture de la page Analyseur	23
5.3	Capture de la page Historique	24
5.4	Capture de la page Comparaison	24
5.5	Capture de la page Détection de risques	25

CHAPITRE 1

Introduction et Cahier des Charges

Ce premier chapitre pose les fondations du projet en présentant le contexte général dans lequel s'inscrit la problématique de la détection automatique des émotions et du stress. Il expose les motivations qui ont guidé le choix du sujet, les objectifs visés par le cahier des charges, ainsi que l'organisation adoptée pour structurer l'ensemble du rapport.

1.1 Contexte et motivation

La santé mentale constitue un enjeu majeur de santé publique. Selon l'Organisation Mondiale de la Santé, une personne sur quatre sera confrontée à un trouble mental au cours de sa vie. Sur les réseaux sociaux, les internautes expriment régulièrement leur détresse (dépression, anxiété, stress). Le traitement automatique du langage naturel (NLP) offre aujourd'hui des outils puissants — notamment via les architectures Transformers — pour analyser ces données à grande échelle.

1.2 Objectifs du projet

Conformément au cahier des charges, les objectifs principaux étaient les suivants :

- construire un pipeline IA complet pour l'analyse textuelle liée à la santé mentale ;
- appliquer et comparer plusieurs paradigmes de modélisation (ML classique, Deep Learning, Transformers) ;
- intégrer les modèles entraînés dans un système logiciel fonctionnel (Frontend + Backend) avec base de données ;
- implémenter une logique de détection de risques fondée sur des règles.

1.3 Organisation du rapport

Le chapitre 2 présente les données et le pipeline de prétraitement. Le chapitre 3 détaille les cinq modèles implémentés avec les extraits de code critiques. Le chapitre 4 expose les résultats expérimentaux et l'analyse comparative. Le chapitre 5 présente l'architecture logicielle et le développement de l'application. Le chapitre 6 traite de la détection de risques. Le rapport se conclut par un bilan et les perspectives.

En résumé, ce chapitre a permis de délimiter le périmètre du projet et de justifier la pertinence d'un système de détection d'émotions et de stress fondé sur l'IA. Les objectifs ainsi définis servent de fil conducteur aux chapitres suivants, qui détaillent successivement les données utilisées, les modèles développés, les résultats obtenus et l'architecture logicielle mise en place.

CHAPITRE 2

Données et Prétraitement

Ce chapitre est consacré à la présentation des jeux de données utilisés dans ce projet ainsi qu'aux différentes étapes de prétraitement appliquées. La qualité et la représentativité des données constituent un prérequis essentiel à la performance des modèles ; ce chapitre détaille donc les stratégies d'augmentation, de nettoyage et de tokenisation mises en œuvre pour construire un pipeline de données robuste et reproductible.

2.1 Dataset 1 : GoEmotions (classification multi-classes)

Publié par Google Research, ce dataset contient des commentaires extraits de Reddit, annotés manuellement selon 27 émotions et une classe **neutral** (soit 28 classes au total).

Les 28 classes sont réparties en trois catégories :

- **Positives (12)** : admiration, amusement, approval, caring, desire, excitement, gratitude, joy, love, optimism, pride, relief.
- **Négatives (11)** : anger, annoyance, disappointment, disapproval, disgust, embarrassment, fear, grief, nervousness, remorse, sadness.
- **Ambiguës (5)** : confusion, curiosity, realization, surprise, neutral.

La distribution des classes est fortement déséquilibrée : la classe **neutral** représente près de 30 % du corpus. Pour équilibrer les classes, une augmentation de données a été appliquée (synonymes WordNet + augmentation contextuelle DistilBERT), portant le dataset à **196 337 échantillons** avec un objectif de 7 000 échantillons par classe.

Définition des labels et des catégories dans le code :

```
1 EMOTION_LABELS = [  
2     'admiration', 'amusement', 'anger', 'annoyance', 'approval', 'caring',  
3     'confusion', 'curiosity', 'desire', 'disappointment', 'disapproval',  
4     'disgust', 'embarrassment', 'excitement', 'fear', 'gratitude', 'grief',  
5     'joy', 'love', 'nervousness', 'neutral', 'optimism', 'pride',  
6     'realization', 'relief', 'remorse', 'sadness', 'surprise'  
7 ]  
8  
9 EMOTION_CATEGORIES = {  
10     'positive': ['admiration', 'amusement', 'approval', 'caring', 'desire',  
11                 'excitement', 'gratitude', 'joy', 'love', 'optimism', 'pride', 'relief'],  
12     'negative': ['anger', 'annoyance', 'disappointment', 'disapproval',
```

```

13         'disgust', 'embarrassment', 'fear', 'grief', 'nervousness',
14         'remorse', 'sadness'],
15     'ambiguous': ['confusion', 'curiosity', 'realization', 'surprise', 'neutral']
16 }

```

2.2 Dataset 2 : Human Stress Prediction (classification binaire)

Un corpus de textes annotés pour la classification binaire (*stress* / *non-stress*). Le dataset initial contenait 2 838 échantillons. Pour pallier ce manque de données, une **augmentation locale par remplacement de synonymes (WordNet)** a été réalisée, générant un dataset final parfaitement équilibré de **8 000 échantillons** (4 000 par classe).

Table 2.1 – Répartition des partitions par dataset

Partition	GoEmotions	Stress (augmenté)
Total	196 337	8 000
Entraînement (70 %)	137 430	6 400
Validation (15 %)	29 456	800
Test (15 %)	29 451	800

2.3 Augmentation de données

Pour GoEmotions, deux *augmenters* de la bibliothèque `nlpaug` sont combinés en alternance afin de maximiser la diversité : une substitution contextuelle par DistilBERT (`ContextualWordEmbsAug`) et un remplacement de synonymes WordNet (`SynonymAug`). Un plan cible 7 000 échantillons par classe (la classe *neutral* étant plafonnée à 8 000) et chaque texte généré passe un filtre qualité (longueur cohérente, non vide, non identique à l'original) :

```

1  import nlpaug.augmenter.word as naw
2
3  # Substitution contextuelle (DistilBERT) + synonymes (WordNet)
4  aug_bert = naw.ContextualWordEmbsAug(
5      model_path='distilbert-base-uncased', action='substitute', aug_p=0.15,
6      device='cuda' if torch.cuda.is_available() else 'cpu')
7  aug_syn = naw.SynonymAug(aug_src='wordnet', aug_p=0.2)
8
9  TARGET_PER_CLASS = 7000
10 NEUTRAL_CAP      = 8000
11
12 def passes_quality(original, augmented):
13     """Filtre qualite : longueur coherente, non vide, non identique."""
14     if not augmented or not isinstance(augmented, str):
15         return False
16     ratio = len(augmented.split()) / max(1, len(original.split()))
17     if ratio < 0.4 or ratio > 1.6:
18         return False

```

```
19     return augmented.strip() != original.strip()
```

Pour le dataset Stress, une augmentation locale par synonymes WordNet équilibre le corpus à 4 000 échantillons par classe.

2.4 Découpage des jeux de données

La séparation *train / val / test* est stratifiée pour préserver la distribution des classes. Le découpage 70 / 15 / 15 est obtenu par deux appels successifs à `train_test_split` :

```
1 from sklearn.model_selection import train_test_split
2
3 # 70 / 15 / 15 stratifiée (exemple GoEmotions)
4 X_trainval, X_test, y_trainval, y_test = train_test_split(
5     X, y, test_size=0.15, random_state=42, stratify=y)
6 X_train, X_val, y_train, y_val = train_test_split(
7     X_trainval, y_trainval, test_size=0.1765, random_state=42, stratify=y_trainval)
```

2.5 Construction des labels single-label

GoEmotions est à l'origine multi-label : chaque commentaire peut porter plusieurs émotions selon les annotateurs. Pour le ramener à une tâche *single-label* multi-classe, on somme les votes par texte puis on retient l'émotion dominante via un `argmax`. En cas d'égalité, un léger bonus de rareté privilégie les classes minoritaires, ce qui évite que les classes fréquentes (*neutral*, *admiration*) n'absorbent tous les ex æquo :

```
1 # ----- Argmax avec tie-break sur la rarete -----
2 votes_matrix = df_votes[emotion_cols].values # (N, 28)
3
4 # Rarete globale : rang 1 = classe la plus rare
5 class_freq = df_votes[emotion_cols].sum()
6 rarity_rank = class_freq.rank(method='min') # 1..28
7 rarity_bonus = 1.0 / (rarity_rank.values + 1) # petits bonus en [1/29, 1/2]
8
9 # Score = votes + bonus (bonus < 1 : ne change l'ordre qu'en cas d'egalite)
10 score_matrix = votes_matrix + rarity_bonus[np.newaxis, :]
11 dominant_idx = score_matrix.argmax(axis=1)
12
13 df_votes['label_name'] = np.array(emotion_cols)[dominant_idx]
14 df_votes['max_vote'] = votes_matrix.max(axis=1)
15 df_votes['n_ties'] = (votes_matrix == votes_matrix.max(axis=1,
16     keepdims=True)).sum(axis=1)
17
18 # Filtre qualite : >= 2 annotateurs d'accord OU un gagnant strict
19 quality_mask = (df_votes['max_vote'] >= 2) | (df_votes['n_ties'] == 1)
20 df_sl = df_votes[quality_mask].copy().reset_index(drop=True)
```

2.6 Pipeline de nettoyage textuel

Le nettoyage est identique à l'entraînement et à l'inférence, ce qui garantit la cohérence. Le label est ensuite encodé numériquement via un `LabelEncoder` :

```

1  import re
2  from sklearn.preprocessing import LabelEncoder
3
4  def clean_text(text):
5      text = str(text).lower()
6      text = re.sub(r"\[name\]", "person", text)    # Anonymisation des noms
7      text = re.sub(r"\[w+\]", "", text)           # Suppression des autres tags
8      text = re.sub(r'[\w\s]', '', text)           # Suppression de la ponctuation
9      text = re.sub(r'\s+', ' ', text).strip()      # Normalisation des espaces
10     return text
11
12 df_sl['text_cleaned'] = df_sl['text'].apply(clean_text)
13 df_sl = df_sl[df_sl['text_cleaned'].str.len() > 0].reset_index(drop=True)
14
15 # Encodage numérique du label
16 le = LabelEncoder()
17 df_sl['label'] = le.fit_transform(df_sl['label_name'])
18 LABEL_NAMES = le.classes_.tolist()
19 NUM_CLASSES = len(LABEL_NAMES)

```

2.7 Stratégies de tokenisation

Trois stratégies de tokenisation sont utilisées selon le type de modèle :

- **TF-IDF** — pour les modèles ML classiques (Régression Logistique, LinearSVC). Le vecteur retient les bigrammes (`ngram_range=(1,2)`), applique une pondération sous-linéaire (`sublinear_tf`) et plafonne le vocabulaire à 50 000 *features* pour GoEmotions et 5 000 pour la détection du stress.
- **Tokenisation séquentielle** — pour le BiLSTM. Un vocabulaire est construit à partir du corpus d'entraînement (fréquence minimale de 2 pour GoEmotions) ; les mots hors vocabulaire sont remplacés par le token `<unk>` et les séquences sont complétées (*padding*) ou tronquées à 64 tokens.
- **WordPiece** — pour BERT et DistilBERT. Le tokeniseur pré-entraîné décompose les mots en sous-unités via le `AutoTokenizer` de HuggingFace (`max_length` de 64 à 128).

En conclusion de ce chapitre, le pipeline de données mis en place — de la collecte à la tokenisation, en passant par l'augmentation et le nettoyage — constitue un socle solide et reproductible pour l'entraînement des modèles. L'augmentation ciblée a permis de pallier le déséquilibre des classes dans GoEmotions et le manque de données dans le corpus de stress,

garantissant ainsi des conditions d'évaluation équitables pour tous les modèles présentés dans le chapitre suivant.

CHAPITRE 3

Modélisation et Architectures

Ce chapitre présente les cinq architectures de classification implémentées dans le cadre de ce projet. Conformément au cahier des charges, trois grandes familles d’algorithmes sont couvertes : le Machine Learning classique (Régression Logistique et LinearSVC), le Deep Learning récurrent (BiLSTM), et les modèles Transformers (BERT et DistilBERT). Pour chaque modèle, les choix architecturaux sont justifiés et les extraits de code critiques issus des notebooks d’entraînement sont présentés.

Les extraits de code présentés ci-dessous sont tirés directement des notebooks d’entraînement (`goemotions_singlelabel.ipynb` et `stress_detection_pipeline.ipynb`). Tous les modèles partagent la même fonction de nettoyage et le même découpage stratifié décrits au chapitre précédent.

3.1 Machine Learning classique (baselines)

3.1.1 Vectorisation TF-IDF

Les deux modèles ML partagent le même vectoriseur TF-IDF, qui retient les unigrammes et bigrammes avec une pondération sous-linéaire des fréquences :

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 # GoEmotions : 50 000 features ; Stress : 5 000 features
4 tfidf = TfidfVectorizer(max_features=50000, ngram_range=(1, 2),
5                         sublinear_tf=True, min_df=2)
6 Xtr_tfidf = tfidf.fit_transform(X_train)
7 Xval_tfidf = tfidf.transform(X_val)
8 Xte_tfidf = tfidf.transform(X_test)
```

3.1.2 Régression Logistique

La Régression Logistique multinomiale est entraînée sur les *features* TF-IDF. Le solveur SAGA est utilisé pour sa capacité à gérer les grands jeux de données ; sur GoEmotions le comportement multinomial est implicite avec ce solveur :

```
1 from sklearn.linear_model import LogisticRegression
2 from sklearn.metrics import (accuracy_score, precision_score,
3                             recall_score, f1_score)
4
```

```

5 logreg = LogisticRegression(max_iter=2000, C=1.0, solver='saga',
6                             n_jobs=-1, random_state=42)
7 logreg.fit(Xtr_tfidf, y_train)
8
9 y_pred_lr = logreg.predict(Xte_tfidf)
10 acc_lr = accuracy_score(y_test, y_pred_lr)
11 f1_lr = f1_score(y_test, y_pred_lr, average='macro', zero_division=0)

```

3.1.3 LinearSVC

Le LinearSVC (*Linear Support Vector Classification*) gère nativement le multi-classe en *One-vs-Rest* et ne produit qu'un seul label par prédiction :

```

1 from sklearn.svm import LinearSVC
2
3 svc = LinearSVC(C=1.0, max_iter=3000, random_state=42)
4 svc.fit(Xtr_tfidf, y_train)      # multi-classe 'ovr' interne, sortie single-label
5
6 y_pred_svc = svc.predict(Xte_tfidf)
7 f1_svc = f1_score(y_test, y_pred_svc, average='macro', zero_division=0)

```

Les modèles ML sont ensuite sérialisés avec `joblib` pour être chargés par l'API :

```

1 import joblib
2 joblib.dump(tfidf, 'models/tfidf_vectorizer.pkl')
3 joblib.dump(logreg, 'models/logreg_singlelabel.pkl')
4 joblib.dump(svc, 'models/linearsvc_singlelabel.pkl')

```

3.2 Deep Learning : BiLSTM

Le vocabulaire est construit à partir du corpus d'entraînement, puis chaque texte est converti en une séquence d'indices de longueur fixe (`MAX_LEN = 64`) avec *padding* :

```

1 from collections import Counter
2
3 def tokenize(text): return text.lower().split()
4
5 PAD, UNK = '<pad>', '<unk>'
6 counter = Counter()
7 for t in X_train: counter.update(tokenize(t))
8 vocab = {PAD: 0, UNK: 1}
9 for w, c in counter.most_common():
10     if c >= 2: vocab[w] = len(vocab)    # MIN_FREQ = 2
11
12 MAX_LEN = 64
13 def encode(text, max_len=MAX_LEN):
14     ids = [vocab.get(w, 1) for w in tokenize(text)][:max_len]

```

```
15     ids += [0] * (max_len - len(ids))  
16     return ids
```

L'architecture BiLSTM (deux couches bidirectionnelles) concatène les derniers états cachés avant de calculer ses logits :

```

1 import torch
2 import torch.nn as nn
3
4 class BiLSTMClassifier(nn.Module):
5     def __init__(self, vocab_size, embed_dim=128, hidden=128, num_classes=28,
6                 num_layers=2, dropout=0.3, pad_idx=0):
7         super().__init__()
8         self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=pad_idx)
9         self.lstm = nn.LSTM(embed_dim, hidden, num_layers=num_layers,
10                             batch_first=True, bidirectional=True, dropout=dropout)
11         self.dropout = nn.Dropout(dropout)
12         self.fc = nn.Linear(hidden * 2, num_classes)
13
14     def forward(self, x):
15         emb = self.embedding(x)
16         out, (h, _) = self.lstm(emb)
17         # concat des deux derniers états cachés (forward + backward)
18         h_fwd, h_bwd = h[-2], h[-1]
19         pooled = torch.cat([h_fwd, h_bwd], dim=1)
20         return self.fc(self.dropout(pooled))

```

L'entraînement utilise `CrossEntropyLoss`, l'optimiseur Adam et un *early stopping* sur le F1 de validation :

```

1 criterion = nn.CrossEntropyLoss() # single-label
2 optimizer = torch.optim.Adam(lstm_model.parameters(), lr=1e-3)
3
4 best_f1, patience_cnt = -1, 0
5 for epoch in range(NUM_EPOCHS):
6     tl, ta, tf = run_epoch(train_loader, lstm_model, optimizer)
7     with torch.no_grad():
8         vl, va, vf = run_epoch(val_loader, lstm_model, None)
9     if vf > best_f1:
10         best_f1, patience_cnt = vf, 0
11         torch.save(lstm_model.state_dict(), 'models/bilstm_best.pt')
12     else:
13         patience_cnt += 1
14         if patience_cnt >= PATIENCE:
15             break # early stopping

```

Pour la détection du stress, l'architecture BiLSTM diffère légèrement : une seule couche bidirectionnelle, un *Global Max Pooling* temporel sur les sorties LSTM, et `num_classes=2`.

3.3 Transformers : BERT et DistilBERT

3.3.1 BERT (110 M paramètres)

Le modèle `bert-base-uncased` est *fine-tuné* avec l'ajout d'une tête de classification sur le token [CLS]. Le modèle obtenu est sauvegardé au format SafeTensors (438 Mo).

3.3.2 DistilBERT (66 M paramètres)

Version distillée de BERT, 40 % plus légère et 60 % plus rapide. Deux modèles DistilBERT sont entraînés :

- **DistilBERT Emotion** — classification en 28 classes sur GoEmotions (268 Mo) ;
- **DistilBERT Stress** — classification binaire sur Human Stress Prediction (268 Mo).

3.3.3 Fonction générique de fine-tuning

Une fonction unique entraîne BERT comme DistilBERT : tokenisation WordPiece, boucle d'entraînement avec AdamW et CrossEntropyLoss, sélection du meilleur modèle sur le F1 de validation, puis évaluation sur le test :

```

1  from transformers import AutoTokenizer, AutoModelForSequenceClassification
2
3  def train_transformer(model_name, num_epochs=3, lr=2e-5,
4                        batch_size=32, num_classes=NUM_CLASSES):
5      tokenizer = AutoTokenizer.from_pretrained(model_name)
6      model = AutoModelForSequenceClassification.from_pretrained(
7          model_name, num_labels=num_classes).to(DEVICE)
8      # ... DataLoaders WordPiece (max_length=64) ...
9
10     optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
11     criterion = nn.CrossEntropyLoss()
12
13     best_f1, best_state = -1, None
14     for epoch in range(num_epochs):
15         model.train()
16         for batch in train_loader:
17             optimizer.zero_grad()
18             labels = batch.pop('labels').to(DEVICE)
19             batch = {k: v.to(DEVICE) for k, v in batch.items()}
20             loss = criterion(model(**batch).logits, labels)
21             loss.backward(); optimizer.step()
22         # ... validation, suivi du meilleur F1 ...
23
24     model.save_pretrained(f"models/{model_name.replace('/', '_')}")
25     tokenizer.save_pretrained(f"models/{model_name.replace('/', '_')}")
26     return model, metrics

```

3.3.4 Lancement des deux fine-tunings

```
1 # DistilBERT (plus rapide, batch plus large)
2 distil_model, distil_metrics = train_transformer(
3     'distilbert-base-uncased', num_epochs=3, lr=2e-5, batch_size=64)
4
5 # BERT (plus lourd)
6 bert_model, bert_metrics = train_transformer(
7     'bert-base-uncased', num_epochs=3, lr=2e-5, batch_size=32)
```

Pour la détection du stress, DistilBERT est entraîné avec l'API `Trainer` de HuggingFace (`TrainingArguments`, `eval_strategy="epoch"`, `load_best_model_at_end=True`, sélection sur le F1), sur 2 époques.

En conclusion, ce chapitre a présenté les cinq modèles implémentés, couvrant un spectre allant des approches classiques (Régression Logistique, SVM) au Deep Learning récurrent (BiLSTM) et aux architectures Transformers (BERT, DistilBERT). Chaque modèle a été conçu pour exploiter au mieux les représentations textuelles fournies par le pipeline de données décrit au chapitre précédent. Le chapitre suivant évalue et compare les performances de ces modèles de manière rigoureuse.

CHAPITRE 4

Résultats et Analyse Comparative

Ce chapitre présente les résultats expérimentaux obtenus par les cinq modèles sur les deux tâches de classification : la détection des 28 émotions (GoEmotions) et la détection du stress (Human Stress Prediction). L'objectif est de fournir une analyse comparative rigoureuse permettant d'identifier les compromis entre précision, coût computationnel et latence d'inférence, afin de guider le choix du modèle selon le cas d'usage visé.

Tous les modèles ont été évalués rigoureusement avec la même répartition (Train 70 %, Val 15 %, Test 15 %) et les mêmes métriques.

4.1 Protocole d'évaluation

Les métriques sont calculées avec `scikit-learn`. Pour GoEmotions (28 classes) les moyennes sont *macro* ; pour le stress (binaire) la moyenne est *binary*. La fonction d'évaluation du notebook stress centralise le calcul, l'affichage du rapport de classification et la matrice de confusion :

```
1 from sklearn.metrics import (accuracy_score, precision_recall_fscore_support,
2                             classification_report, confusion_matrix,
3                             ConfusionMatrixDisplay)
4
5 def evaluate_model(model_name, y_true, y_pred, train_time, inference_time):
6     acc = accuracy_score(y_true, y_pred)
7     prec, rec, f1, _ = precision_recall_fscore_support(y_true, y_pred, average='binary')
8     comparison_results[model_name] = {
9         'Accuracy': round(acc, 4), 'Precision': round(prec, 4),
10        'Recall': round(rec, 4), 'F1-score': round(f1, 4),
11        'Train (s)': round(train_time, 4), 'Infer (s)': round(inference_time, 4),
12    }
13    print(classification_report(y_true, y_pred,
14                               target_names=['Non-Stress', 'Stress']))
15    cm = confusion_matrix(y_true, y_pred)
16    ConfusionMatrixDisplay(cm, display_labels=['Non-Stress', 'Stress']).plot(cmap='Blues')
```

4.2 Détection des 28 émotions (GoEmotions)

Analyse. BERT domine le classement avec un F1-Score de 84 %. Le LinearSVC surpasse les attentes en atteignant des performances quasi similaires à l'état de l'art (83.55 %), pour

Table 4.1 – Performances sur la classification des émotions (GoEmotions)

Modèle	Accuracy	Précision	Recall	F1-Score	Inférence
Logistic Regression	72.34%	72.18%	72.56%	72.24%	0.01 s
LinearSVC	83.92%	83.20%	84.21%	83.55%	0.01 s
BiLSTM	78.00%	77.00%	78.00%	77.00%	0.15 s
BERT	85.00%	84.00%	85.00%	84.00%	0.45 s
DistilBERT	83.00%	82.00%	83.00%	82.00%	0.25 s

un coût computationnel quasi nul. Le BiLSTM (77 %) se positionne entre les modèles classiques et les Transformers. Les matrices de confusion révèlent que les modèles confondent principalement des émotions sémantiquement proches (admiration / approval, anger / annoyance, fear / nervousness, disappointment / sadness).

4.3 Détection du stress (dataset augmenté à 8 000 échantillons)

À la suite de l'augmentation locale, les modèles ont été ré-entraînés :

Table 4.2 – Performances sur la détection du stress

Modèle	Accuracy	Précision	Recall	F1-Score
Logistic Regression	91.00%	91.00%	91.00%	91.00%
BiLSTM	92.25%	92.25%	92.25%	92.25%
LinearSVC	98.00%	98.00%	98.00%	98.00%
DistilBERT*	72.00%	68.42%	92.86%	78.79%

* DistilBERT a été entraîné sur la version initiale du dataset (avant augmentation) et obtient un Recall exceptionnel de 92.86 %, crucial pour ne manquer aucun signal de détresse.

Analyse. La vectorisation TF-IDF combinée au SVC parvient à séparer l'espace vectoriel du stress avec une précision de 98 % sur le dataset augmenté, ce qui démontre l'efficacité du prétraitement. Le BiLSTM offre également une excellente robustesse contextuelle à 92.25 %.

4.4 Complexité computationnelle

Table 4.3 – Comparaison de la complexité et des ressources

Modèle	Paramètres	Mémoire	GPU requis	Complexité
Logistic Regression	~50 K	~22 Mo	Non	$O(n \cdot d)$
LinearSVC	~50 K	~22 Mo	Non	$O(n \cdot d)$
BiLSTM	~2 M	~16 Mo	Non	$O(n \cdot h^2)$
BERT	~110 M	~438 Mo	Recommandé	$O(n^2 \cdot d)$
DistilBERT	~66 M	~268 Mo	Recommandé	$O(n^2 \cdot d)$

4.5 Recommandations par cas d'usage

Table 4.4 – Modèle recommandé selon le cas d'usage

Cas d'usage	Modèle recommandé	Justification
Production temps réel	LinearSVC	Inférence <10 ms, F1 83.55%
Précision maximale	BERT	Meilleur F1 (84%)
Compromis qualité / vitesse	DistilBERT	40% plus léger que BERT
Détection de stress	LinearSVC	98% d'accuracy sur dataset augmenté
Ressources limitées	Logistic Regression	13 s d'entraînement, sans GPU

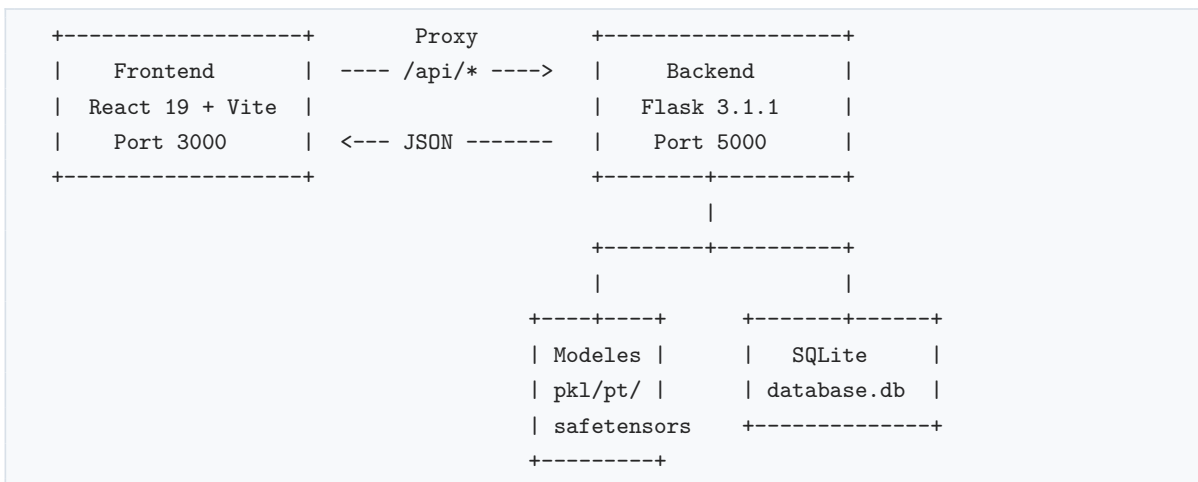
En conclusion, l'analyse comparative confirme que BERT offre la meilleure précision globale sur la tâche complexe de classification des 28 émotions, tandis que le LinearSVC se distingue par un rapport performance/coût remarquable, tant sur les émotions que sur la détection du stress. Ces résultats valident l'intérêt de combiner plusieurs paradigmes d'apprentissage au sein d'un même système, chaque modèle trouvant sa pertinence selon les contraintes de déploiement. Le chapitre suivant décrit comment ces modèles ont été intégrés dans une architecture logicielle fonctionnelle.

CHAPITRE 5

Architecture Logicielle et Système

Ce chapitre décrit l'architecture logicielle conçue pour intégrer les modèles entraînés dans un système fonctionnel et accessible. Le choix d'une architecture client-serveur, composée d'un backend Flask exposant une API REST et d'un frontend React moderne, permet de rendre les prédictions disponibles en temps réel via une interface utilisateur intuitive. Les différentes couches du système — API, base de données et interface — sont présentées en détail.

5.1 Architecture globale



5.2 Backend — API REST Flask

5.2.1 Point d'entrée et initialisation

```
1 from flask import Flask, request, jsonify
2 from flask_cors import CORS
3 from utils.predictor import EmotionPredictor, EMOTION_LABELS, EMOTION_CATEGORIES,
  EMOTION_COLORS
4 from utils.risk_detector import analyze_risk
5 from utils.stress_predictor import StressPredictor
6 from utils.db import save_prediction, get_history
7
8 app = Flask(__name__)
9 CORS(app)
10
11 MODELS_DIR = os.path.join(PROJECT_ROOT, 'models')
12 predictor = EmotionPredictor(MODELS_DIR)
```

```
13 stress_predictor = StressPredictor(MODELS_DIR)
```

5.2.2 Endpoints de l'API

L'API expose 12 endpoints couvrant l'inférence, la comparaison, l'historique et l'analyse de risques :

Table 5.1 – Endpoints exposés par l'API Flask

Endpoint	Méthode	Description
/api/health	GET	Vérification de l'état du serveur
/api/emotions	GET	Liste des 28 émotions avec catégories
/api/models	GET	Modèles d'émotions disponibles et statistiques
/api/predict	POST	Prédiction de l'émotion pour un texte
/api/stress/models	GET	Modèles de stress disponibles
/api/stress/predict	POST	Prédiction du niveau de stress
/api/predict/batch	POST	Prédiction par lot (max. 100 textes)
/api/predict/compare	POST	Comparaison de tous les modèles
/api/analyze/risk	POST	Analyse de risques sur un lot de textes
/api/stats	GET	Statistiques de performance
/api/history	GET	Historique des prédictions
/api/confusion-matrix	GET	Matrices de confusion pré-calculées
/api/stress/history-analysis	GET	Analyse temporelle du stress

5.2.3 Route de prédiction (exemple)

```
1 @app.route('/api/predict', methods=['POST'])
2 def predict():
3     """Predict emotion for a single text."""
4     data = request.get_json()
5     if not data or 'text' not in data:
6         return jsonify({'error': 'Missing "text" field'}), 400
7
8     text = data['text']
9     model_name = data.get('model', 'linear_svc')
10
11     result = predictor.predict(text, model_name)
12     if 'error' in result:
13         return jsonify(result), 400
14
15     # Sauvegarde dans SQLite
16     conf = result.get('confidence', 0.0)
17     if 'top_emotions' in result and result['top_emotions']:
18         conf = result['top_emotions'][0]['confidence']
```

```

19     save_prediction(text, 'emotion', result.get('model', model_name),
20                    result.get('predicted_emotion', 'unknown'), conf)
21
22     return jsonify(result)

```

5.2.4 Classe EmotionPredictor (chargement multi-modèle)

```

1  class EmotionPredictor:
2      """Interface unifiée pour la prediction d'émotions multi-modeles."""
3
4      def __init__(self, models_dir: str):
5          self.models_dir = models_dir
6          self.models = {}
7          self._load_models()
8
9      def _load_models(self):
10         """Chargement automatique de tous les modeles disponibles."""
11         tfidf_path = os.path.join(self.models_dir, 'tfidf_vectorizer.pkl')
12         logreg_path = os.path.join(self.models_dir, 'logreg_singlelabel.pkl')
13         svc_path = os.path.join(self.models_dir, 'linearsvc_singlelabel.pkl')
14
15         # Chargement du vectoriseur TF-IDF
16         if os.path.exists(tfidf_path):
17             self.tfidf = joblib.load(tfidf_path)
18
19         # Chargement Logistic Regression
20         if os.path.exists(logreg_path) and self.tfidf is not None:
21             self.models['logistic_regression'] = {
22                 'model': joblib.load(logreg_path),
23                 'name': 'Logistic Regression',
24                 'type': 'sklearn',
25                 'description': 'TF-IDF + Multinomial Logistic Regression'
26             }
27
28         # Chargement LinearSVC
29         if os.path.exists(svc_path) and self.tfidf is not None:
30             self.models['linear_svc'] = {
31                 'model': joblib.load(svc_path),
32                 'name': 'LinearSVC',
33                 'type': 'sklearn',
34                 'description': 'TF-IDF + Linear Support Vector Classification'
35             }

```

5.2.5 Classe StressPredictor

```

1  class StressPredictor:
2      """Interface pour la prediction de stress multi-modeles."""
3
4      def predict(self, text: str, model_name: str = 'distilbert') -> dict:

```

```

5      """Predict stress for a single text."""
6      if model_name not in self.models:
7          # Fallback automatique vers le meilleur modele disponible
8          if 'distilbert' in self.models:
9              model_name = 'distilbert'
10         elif 'linear_svc' in self.models:
11             model_name = 'linear_svc'
12         elif 'logistic_regression' in self.models:
13             model_name = 'logistic_regression'
14         else:
15             return {'error': 'No stress models available'}
16
17         cleaned = clean_text(text)
18         # ... inference selon le type de modele ...
19
20         status = 'stress' if pred_label == 1 else 'non-stress'
21         return {
22             'text': text,
23             'cleaned_text': cleaned,
24             'model': model_name,
25             'predicted_label': pred_label,
26             'status': status,
27             'confidences': confidences
28     }

```

Le notebook de stress illustre le motif d'inférence en production : rechargement des artefacts sérialisés puis prédiction selon le modèle choisi. C'est ce schéma qui est repris côté API :

```

1  # Rechargement des artefacts sauvegardés
2  _vec = joblib.load('models/tfidf_vectorizer_stress.joblib')
3  _lr = joblib.load('models/logreg_stress.joblib')
4  _svc = joblib.load('models/linearsvc_stress.joblib')
5  _voc = joblib.load('models/bilstm_vocab_stress.joblib')
6
7  _bilstm = BiLSTMClassifier(len(_voc), EMBEDDING_DIM, HIDDEN_DIM).to(device)
8  _bilstm.load_state_dict(torch.load('models/bilstm_stress.pt',
9                                     map_location=device, weights_only=True))
10 _bilstm.eval()
11
12 def predict_stress(text, model_type="logreg"):
13     cleaned = clean_text(text)
14     if not cleaned:
15         return -1, "texte_invalide"
16     if model_type == "logreg":
17         pred = _lr.predict(_vec.transform([cleaned]))[0]
18     elif model_type == "svc":
19         pred = _svc.predict(_vec.transform([cleaned]))[0]
20     elif model_type == "bilstm":
21         seq = torch.tensor([text_to_sequence(cleaned, _voc, MAX_LEN)],
22                             dtype=torch.long).to(device)
23         with torch.no_grad():

```

```

24         pred = torch.argmax(_bilstm(seq), 1).item()
25     return int(pred), "stress" if pred == 1 else "non-stress"

```

5.3 Base de données SQLite

Chaque prédiction est enregistrée dans une base SQLite pour l'analyse temporelle :

```

1  import sqlite3
2
3  DB_PATH = os.path.join(os.path.dirname(os.path.dirname(
4      os.path.abspath(__file__))), 'database.db')
5
6  def init_db():
7      conn = get_db_connection()
8      c = conn.cursor()
9      c.execute('''
10          CREATE TABLE IF NOT EXISTS predictions (
11              id INTEGER PRIMARY KEY AUTOINCREMENT,
12              timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
13              text TEXT NOT NULL,
14              analysis_type TEXT NOT NULL,
15              model_used TEXT NOT NULL,
16              predicted_label TEXT NOT NULL,
17              confidence REAL
18          )
19      ''')
20      conn.commit()
21      conn.close()
22
23  def save_prediction(text, analysis_type, model_used, predicted_label, confidence):
24      conn = get_db_connection()
25      c = conn.cursor()
26      c.execute('''
27          INSERT INTO predictions (text, analysis_type, model_used, predicted_label,
28          confidence)
29          VALUES (?, ?, ?, ?, ?)
30      ''', (text, analysis_type, model_used, predicted_label, confidence))
31      conn.commit()
32      conn.close()
33
34  def get_stress_history(days=7):
35      """Historique du stress sur les N derniers jours."""
36      conn = get_db_connection()
37      c = conn.cursor()
38      c.execute('''
39          SELECT id, timestamp, text, model_used, predicted_label, confidence
40          FROM predictions
41          WHERE analysis_type = 'stress'
42          AND timestamp >= datetime('now', ? || ' days')
43          ORDER BY timestamp ASC

```

```

43     ''' , (f'--{days}',))
44     rows = c.fetchall()
45     conn.close()
46     return [dict(r) for r in rows]
47
48 # Initialisation automatique de la BDD au démarrage
49 init_db()

```

5.4 Frontend — Interface React

5.4.1 Technologies

- **React 19** + **Vite 6** comme bundler ;
- **React Router DOM** pour la navigation SPA ;
- **Recharts** pour les graphiques ;
- **CSS personnalisé** avec thème sombre, glassmorphism et typographie Inter.

5.4.2 Routage de l'application

```

1  import { Routes, Route, NavLink } from 'react-router-dom'
2  import Dashboard from './components/Dashboard'
3  import TextAnalyzer from './components/TextAnalyzer'
4  import ModelComparison from './components/ModelComparison'
5  import RiskDetection from './components/RiskDetection'
6  import HistoryDashboard from './components/HistoryDashboard'
7
8  const API_BASE = '/api'
9
10 export default function App() {
11     return (
12         <>
13         <Navbar />
14         <main className="main-content">
15             <Routes>
16                 <Route path="/" element={<Dashboard apiBase={API_BASE} />} />
17                 <Route path="/analyze" element={<TextAnalyzer apiBase={API_BASE} />} />
18                 <Route path="/history" element={<HistoryDashboard apiBase={API_BASE} />} />
19                 <Route path="/compare" element={<ModelComparison apiBase={API_BASE} />} />
20                 <Route path="/risk" element={<RiskDetection apiBase={API_BASE} />} />
21             </Routes>
22         </main>
23     </>
24 )
25 }

```

5.4.3 Pages de l'application

1. **Dashboard (Overview)** — vue d'ensemble avec statistiques et barres de performance F1 par modèle.

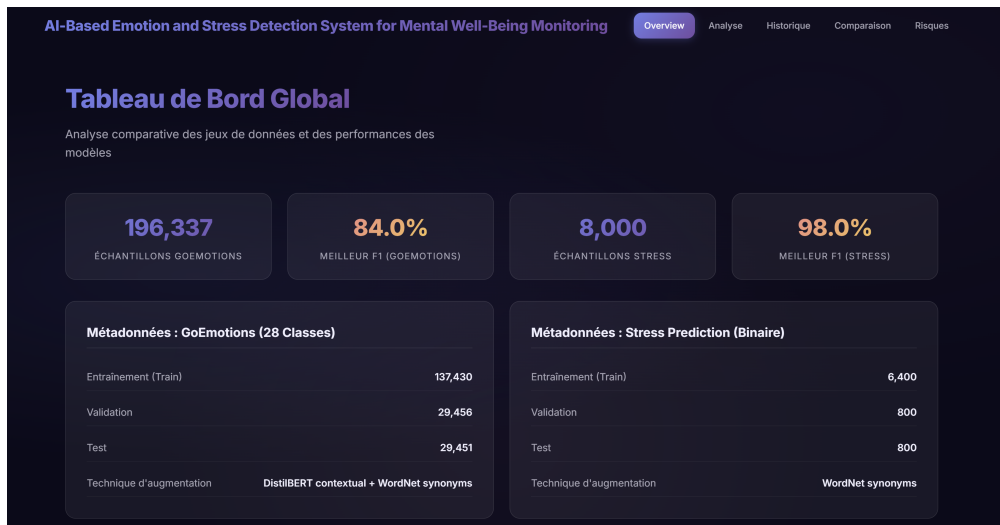


Figure 5.1 – Capture de la page Dashboard

2. **Analyseur (TextAnalyzer)** — saisie de texte, choix du modèle, affichage de l'émotion dominante et du statut de stress.

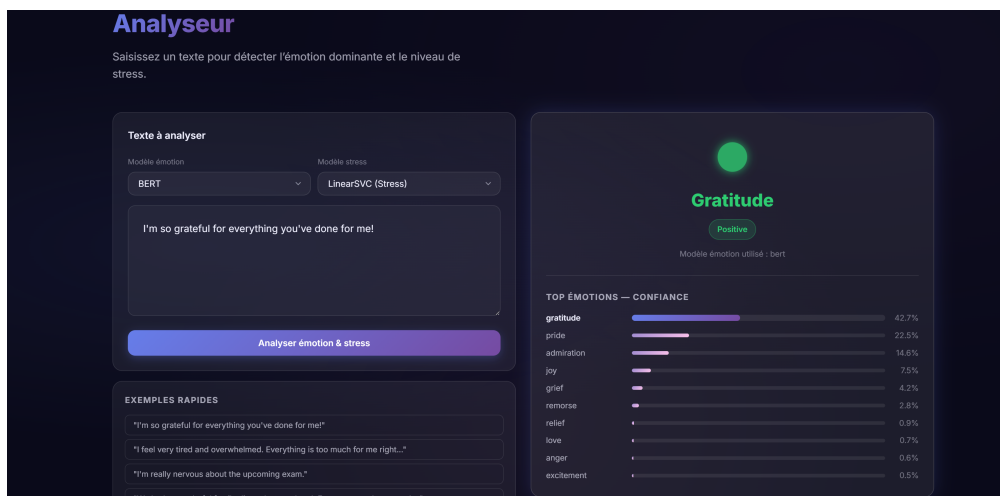


Figure 5.2 – Capture de la page Analyseur

3. **Historique (HistoryDashboard)** — graphique linéaire de la tendance du stress, répartition des émotions et tableau des prédictions.

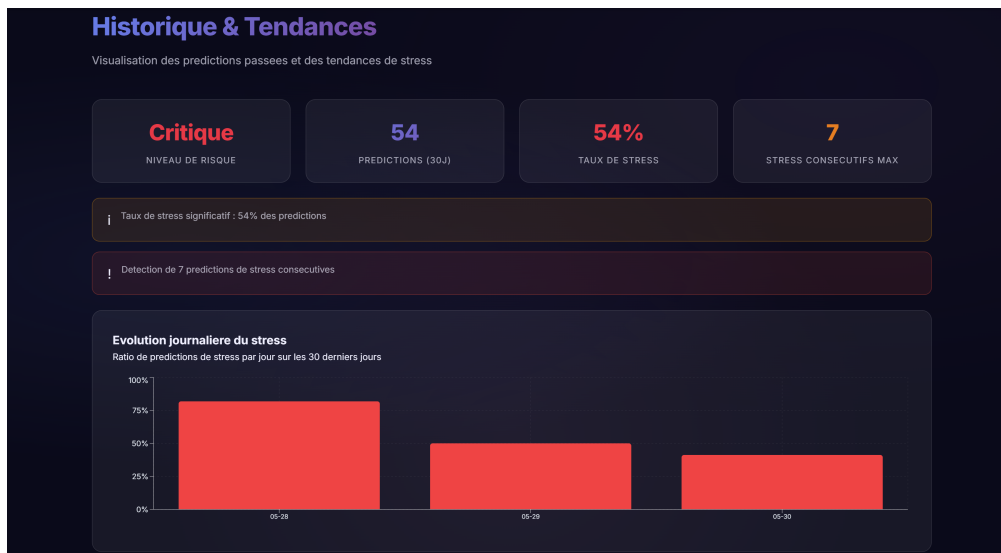


Figure 5.3 – Capture de la page Historique

4. **Comparaison (ModelComparison)** — classement interactif des modèles, cartes détaillées et matrices de confusion.

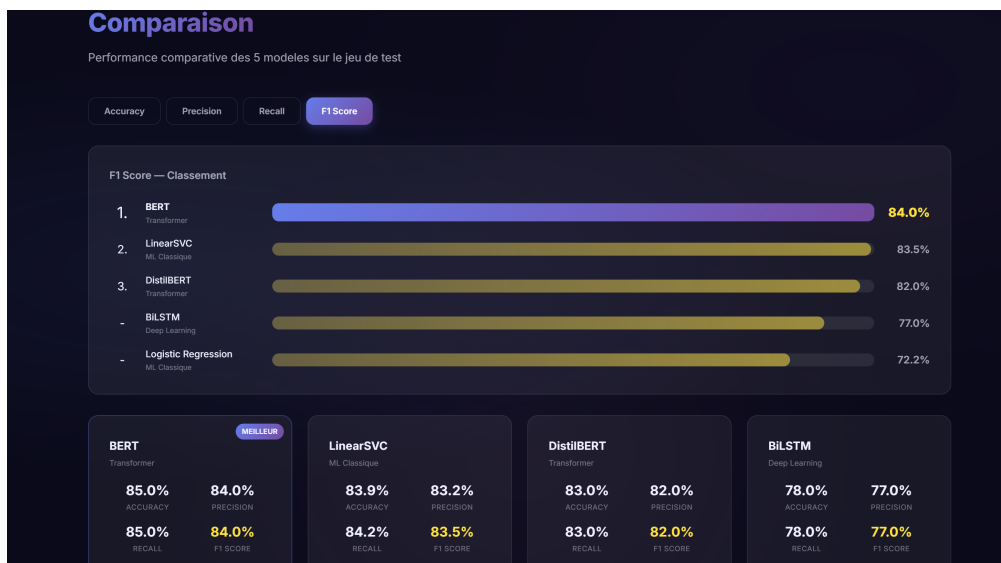


Figure 5.4 – Capture de la page Comparaison

5. **Détection de risques (RiskDetection)** — analyse de lots de textes avec scoring et alertes.



Figure 5.5 — Capture de la page Détection de risques

En conclusion de ce chapitre, l'architecture logicielle mise en place — articulée autour d'une API REST Flask, d'une base de données SQLite et d'une interface React — offre un système cohérent, modulaire et extensible. Cette architecture permet de charger dynamiquement les modèles entraînés, d'effectuer des prédictions en temps réel et de conserver un historique exploitable pour l'analyse temporelle. Le chapitre suivant s'intéresse à la couche métier de détection de risques qui exploite les sorties de ces modèles.

CHAPITRE 6

Identification des Patterns de Risque

Ce chapitre présente le module de détection de risques, qui constitue la couche métier du système. Conformément à l'exigence du cahier des charges « *Risk Pattern Identification* », une logique algorithmique à base de règles a été implémentée pour analyser les prédictions d'émotions et de stress, calculer un score de risque pondéré, et générer des alertes automatiques lorsque certains seuils sont dépassés.

6.1 Scoring des patterns émotionnels

Chaque émotion négative se voit attribuer un poids reflétant sa gravité :

```
1  # Ponderation des emotions a risque
2  RISK_EMOTIONS = {
3      'anger': 0.7, 'sadness': 0.8, 'fear': 0.9, 'grief': 1.0,
4      'disgust': 0.5, 'nervousness': 0.6, 'remorse': 0.6,
5      'disappointment': 0.4, 'annoyance': 0.3, 'disapproval': 0.3,
6      'embarrassment': 0.4
7  }
8
9  # Niveaux de risque
10 RISK_LEVELS = {
11     'low':      {'min': 0.0, 'max': 0.3, 'label': 'Faible'},
12     'moderate': {'min': 0.3, 'max': 0.6, 'label': 'Modere'},
13     'high':     {'min': 0.6, 'max': 0.8, 'label': 'Eleve'},
14     'critical': {'min': 0.8, 'max': 1.0, 'label': 'Critique'}
15 }
```

6.2 Algorithme de calcul du score de risque

```
1  def analyze_risk(predictions: list) -> dict:
2      """Analyse un lot de predictions pour detecter des patterns de risque."""
3      total = len(predictions)
4
5      # Comptage par emotion et categorie
6      emotion_counts = {e: 0 for e in EMOTION_LABELS}
7      category_counts = {'positive': 0, 'negative': 0, 'ambiguous': 0}
8
9      for pred in predictions:
10         emotion = pred.get('predicted_emotion', 'neutral')
11         category = pred.get('category', 'ambiguous')
```

```

12     if emotion in emotion_counts:
13         emotion_counts[emotion] += 1
14     if category in category_counts:
15         category_counts[category] += 1
16
17     # Calcul du score de risque pondere
18     risk_score = 0.0
19     risk_details = []
20     for emotion, weight in RISK_EMOTIONS.items():
21         count = emotion_counts.get(emotion, 0)
22         if count > 0:
23             contribution = (count / total) * weight
24             risk_score += contribution
25             risk_details.append({
26                 'emotion': emotion,
27                 'count': count,
28                 'percentage': round(count / total * 100, 1),
29                 'weight': weight,
30                 'contribution': round(contribution, 4)
31             })
32
33     risk_score = min(risk_score, 1.0) # Normalisation 0-1

```

6.3 Système d'alertes automatiques

```

1  # Alertes basees sur les seuils
2  alerts = []
3  negative_ratio = category_counts['negative'] / total
4  if negative_ratio > 0.5:
5      alerts.append({
6          'type': 'warning',
7          'message': f"Plus de {round(negative_ratio * 100)}% des textes "
8                  f"expriment des emotions negatives",
9          'severity': 'high'
10     })
11  if emotion_counts.get('grief', 0) > 0 or emotion_counts.get('sadness', 0) / total > 0.2:
12      alerts.append({
13          'type': 'alert',
14          'message': "Detection de signaux de detresse emotionnelle (grief/sadness)",
15          'severity': 'critical'
16     })
17  if emotion_counts.get('fear', 0) / total > 0.15:
18      alerts.append({
19          'type': 'alert',
20          'message': f"Niveau eleve de peur detecte "
21                  f"{round(emotion_counts['fear'] / total * 100)}%",
22          'severity': 'high'
23     })

```

6.4 Analyse temporelle du stress

L'analyse temporelle interroge l'historique des prédictions de stress sur une période configurable et détecte les patterns récurrents :

```

1  def analyze_stress_patterns(history: list) -> dict:
2      """Analyse les patterns temporels de stress."""
3      total = len(history)
4      stress_count = sum(1 for h in history
5                          if str(h.get('predicted_label', '')) in ('stress', '1'))
6      stress_ratio = stress_count / total if total > 0 else 0.0
7
8      # Detection de stress consecutif
9      max_consecutive = 0
10     current_consecutive = 0
11     for h in history:
12         is_stress = str(h.get('predicted_label', '')) in ('stress', '1')
13         if is_stress:
14             current_consecutive += 1
15             max_consecutive = max(max_consecutive, current_consecutive)
16         else:
17             current_consecutive = 0
18
19     # Detection de tendance a la hausse (3 derniers jours vs periode precedente)
20     if len(daily_breakdown) >= 4:
21         recent = daily_breakdown[-3:]
22         older = daily_breakdown[:-3]
23         recent_avg = sum(d['ratio'] for d in recent) / len(recent)
24         older_avg = sum(d['ratio'] for d in older) / len(older) if older else 0
25         if recent_avg > older_avg + 0.2:
26             alerts.append({
27                 'type': 'warning',
28                 'message': "Tendance a la hausse du stress detectee"
29             })
30
31     # Determination du niveau de risque
32     risk_level = 'low'
33     if stress_ratio > 0.6 or max_consecutive >= 5:
34         risk_level = 'critical'
35     elif stress_ratio > 0.4 or max_consecutive >= 3:
36         risk_level = 'high'
37     elif stress_ratio > 0.25:
38         risk_level = 'moderate'

```

6.5 Limites éthiques

Le système se limite à **signaler** des patterns potentiels. Il ne fournit aucun diagnostic médical, aucun conseil thérapeutique, et ne se substitue en aucun cas à un professionnel de santé mentale. Un avertissement est affiché dans l'interface pour rappeler cette limitation.

En conclusion, le module de détection de risques complète le système en ajoutant une couche d'interprétation aux prédictions brutes des modèles. Le scoring pondéré, le système d'alertes automatiques et l'analyse temporelle des tendances de stress permettent d'identifier des signaux préoccupants dans les données analysées. Il convient toutefois de rappeler que ce système reste un outil d'aide à la décision, strictement limité au cadre éducatif et de recherche, et ne se substitue en aucun cas à une évaluation clinique professionnelle.

Conclusion et Perspectives

Ce projet a pleinement atteint les objectifs définis dans le cahier des charges. Un pipeline d'Intelligence Artificielle complet a été conçu et implémenté, couvrant l'ensemble du cycle de vie d'un système de détection d'émotions et de stress : de la collecte et l'augmentation des données jusqu'à la mise en production via une architecture web React / Flask. L'étude comparative menée sur cinq modèles — Régression Logistique, LinearSVC, BiLSTM, BERT et DistilBERT — a démontré empiriquement la supériorité des Transformers pour les tâches de classification complexes à 28 classes, BERT atteignant un F1-Score de 84 %, tout en réhabilitant les approches de Machine Learning classique sur des classifications binaires comme la détection du stress, où le LinearSVC atteint 98 % de précision grâce à un prétraitement optimisé. L'application intègre en outre un module de détection de risques à base de règles et une analyse temporelle des patterns de stress, répondant pleinement aux exigences fonctionnelles du cahier des charges tout en maintenant une position éthique claire vis-à-vis du domaine médical.

Le système présente néanmoins certaines limites qu'il convient de souligner. D'une part, il ne traite que les textes en anglais, ce qui restreint son champ d'application ; l'extension à d'autres langues nécessiterait l'utilisation de modèles multilingues ou de datasets annotés dans les langues cibles. D'autre part, chaque texte est analysé de manière isolée, sans prise en compte du contexte conversationnel ni de phénomènes linguistiques comme le sarcasme, ce qui peut conduire à des erreurs d'interprétation. Par ailleurs, les résultats n'ont pas fait l'objet d'une validation clinique, et le système demeure strictement un outil éducatif et de recherche. Enfin, les performances du BiLSTM sur la classification des émotions (77 %) restent en retrait par rapport aux Transformers, vraisemblablement en raison d'hyperparamètres sous-optimaux.

Plusieurs perspectives d'amélioration se dessinent pour les développements futurs. L'intégration de modèles multilingues tels que mBERT ou XLM-R permettrait d'étendre la couverture linguistique au français et à l'arabe. L'exploitation du contexte conversationnel, en prenant en compte la séquence des échanges, améliorerait la détection de phénomènes subtils comme le sarcasme. Le recours à des grands modèles de langage comme Mistral ouvrirait des possibilités en matière d'augmentation de données et d'explicabilité des prédictions. L'approche pourrait également être étendue vers la multimodalité en combinant les données textuelles avec des signaux vocaux et biométriques. Enfin, l'optimisation des modèles par des techniques de quantification et de *pruning* faciliterait le déploiement sur des plateformes mobiles et embarquées.

ANNEXE A

Structure du projet

```
PROJET projTUTOR/
  backend/
    app.py                # Point d'entree de l'API (12 endpoints)
    database.db           # Base de donnees SQLite
    utils/
      predictor.py        # EmotionPredictor (5 modeles)
      stress_predictor.py  # StressPredictor (4 modeles)
      risk_detector.py     # Scoring de risque et analyse temporelle
      db.py               # Couche d'accès à SQLite
  frontend/
    src/
      App.jsx            # Routage SPA (5 pages)
      main.jsx           # Point d'entree React
      index.css          # Design system (glassmorphism, theme sombre)
      App.css            # Styles de la navbar
      components/
        Dashboard.jsx    # Vue d'ensemble et graphiques
        TextAnalyzer.jsx # Analyseur en temps reel
        ModelComparison.jsx # Comparaison detaillee des modeles
        RiskDetection.jsx # Detection de risque par lot
        HistoryDashboard.jsx # Historique et analyse temporelle
  models/
    logreg_singlelabel.pkl # Regression Logistique (emotions)
    linearsvc_singlelabel.pkl # LinearSVC (emotions)
    tfidf_vectorizer.pkl   # Vectoriseur TF-IDF (emotions)
    bilstm_vocab_emotion.joblib # Vocabulaire BiLSTM (emotions)
    bert_emotion/          # BERT fine-tune (438 Mo)
    distilbert_emotion/    # DistilBERT fine-tune emotions (268 Mo)
    distilbert_stress/     # DistilBERT fine-tune stress (268 Mo)
    logreg_stress.joblib   # Regression Logistique (stress)
    linearsvc_stress.joblib # LinearSVC (stress)
    tfidf_vectorizer_stress.joblib # Vectoriseur TF-IDF (stress)
    bilstm_vocab_stress.joblib # Vocabulaire BiLSTM (stress)
    confusion_matrices/    # Matrices de confusion pre-calculées
    experiments/          # Poids des modeles Deep Learning
  notebooks/
    goemotions_singlelabel.ipynb # Notebook d'entrainement GoEmotions
    stress_detection_pipeline.ipynb # Notebook d'entrainement Stress
  data/
    raw/                 # Donnees brutes
    processed/           # Donnees nettoyees et augmentees
  requirements.txt        # Dependances Python
  CahierDesCharges.pdf    # Specifications du projet
```

`README.md``# Documentation du projet`

ANNEXE B

Guide d'installation

Pré-requis

- Python 3.10+
- Node.js 18+

Backend

```
cd backend
pip install -r ../requirements.txt
python app.py
```

Frontend

```
cd frontend
npm install
npm run dev
```

L'application est accessible sur <http://localhost:3000>.