Projet NLP : “Chercheur des talents”

Objectif général

Créer un système intelligent capable de **détecter automatiquement les meilleurs talents** (dans une entreprise ou sur une plateforme professionnelle) puis de **leur proposer des offres d’emploi adaptées**, même s’ils n’en ont pas fait la demande.

C’est un système qui **automatise le sourcing**, facilite la **concurrence entre entreprises** et améliore le **matching candidats–offres**.

Comment ce projet devient un vrai projet NLP ?

Pour faire ça, le cœur du projet repose sur **l’analyse automatique du langage** :

✔️ 1. Extraction d’informations (NLP – Information Extraction)

Ton système doit pouvoir analyser :

- CV
- profils LinkedIn
- rapports de performance internes
- mails professionnels
- contributions GitHub / textes / publications
- etc.

Et extraire :

- compétences techniques (“Python”, “DevOps”, “NLP”...)

- soft skills (“leadership”, “communication”)
- niveaux
- expériences
- projets réalisés
- domaines d’expertise

👉 Modèles : SpaCy, Transformers (BERT, RoBERTa), NER, keyword extraction.

✔️ 2. Évaluation de la qualité des talents (Ranking NLP)

Le système doit **classer les candidats** selon :

- pertinence des compétences
- niveau d’expérience
- performance dans des projets
- indicateurs internes

👉 Techniques NLP :

- embeddings (Sentence-BERT)
- scoring / ranking
- similarity search (FAISS)

✔️ 3. Matching Talent–Offre d’emploi

Comparer ce que dit :

- la **description de l'offre**
- le **profil du candidat**

Déterminer :

- taux de compatibilité
- compétences manquantes
- compétences clés
- postes les plus adaptés

👉 NLP : semantic similarity, BERT fine-tuning, classification multi-label.

✔️ 4. Recommandation proactive (Recommender System + NLP)

Le système **n'attend pas** que le candidat postule.
Il détecte :

- profils rares
- talents performants
- compétences recherchées

Puis **propose automatiquement** des offres pertinentes.

👉 Techniques :

- recommender systems
- embeddings similarity
- clustering KMeans + NLP

Architecture simple du projet

Voici une architecture que tu peux présenter :

1) Collecte de données

CV, profils, descriptions de postes…

2) Prétraitement NLP

Tokenization, lemmatisation, nettoyage.

3) Extraction des compétences

NER + keywords extraction.

4) Embedding des documents

Sentence-BERT ou autres modèles transformer.

5) Matching / Similarité

FAISS similarity search.

6) Recommandation automatique

Algorithme de ranking.

7) Interface

Dashboard permettant :

- recherche de talents
- suggestions automatiques d’offres

Projet NLP — « Chercheur des talents » : dossier complet

> Document complet fourni "un par un" : problématique scientifique, plan détaillé, architecture, jeu de données, code modèle, et contenu pour une présentation PowerPoint.

1) Problématique scientifique

Contexte

Les entreprises dépensent beaucoup de temps et d'argent pour sourcer des candidats. Un système automatisé peut repérer des talents (internes ou externes) à partir de textes (CV, profils, rapports, contributions) et proposer proactivement des offres.

Objectif général

Construire un pipeline NLP capable de détecter, scorer et recommander automatiquement les meilleurs talents pour des offres d'emploi, sans action explicite du candidat.

Questions de recherche

- Peut-on définir un score robuste de "talent" basé uniquement sur des données textuelles hétérogènes ?
- Quels signaux textuels (compétences, impact projet, leadership, publications) prédisent le plus la qualité d'un talent ?
- Quelle architecture (embeddings + index vecteur vs fine-tuned classifier) donne le meilleur compromis précision/latence pour le matching proactif ?
- Comment limiter les biais (genre, âge, origine) dans l'identification et le ranking ?

Hypothèses

1. Les embeddings de phrases (Sentence-BERT) combinés à des features structurés (années d'expérience, poste) permettent un ranking de haute qualité.
2. Les signaux d'impact (projets réussis, contributions open-source, publications) sont corrélés avec un score de talent supérieur.
3. Une étape de rééquilibrage et de post-traitement réduit significativement le biais systémique.

Indicateurs de succès

- Precision@10 et Recall@K sur un jeu d'évaluation annoté.
- AUC pour une tâche binaire (talent / non talent) si disponible.
- Évaluation humaine (panel RH) : taux d'acceptation des recommandations > 70%.
- Latence de recherche < 200 ms pour 1M de profils (objectif scale).

2) Plan complet (work breakdown & milestones)

Phase 0 — Spécification (1–2 semaines)

- Définir format des sources (CV, LinkedIn, rapports internes).
- Définir évaluation: quelles annotations, quel panel RH.
- Définir KPIs.

Phase 1 — Collecte et prétraitement (2–4 semaines)

- Récupération des données (scraping, ingestion interne).
- Normalisation (texte brut, suppression PII si besoin, anonymisation pour tests).
- Extraction initiale: NER compétences, postes, dates.

Livrables : dataset nettoyé + scripts de preprocessing.

Phase 2 — Feature & Representation (2–4 semaines)

- Extraire features textuels (skills list, projets, verbatims).
- Générer embeddings (Sentence-BERT / multilingual si multi-langues).
- Construire features supplémentaires (TF-IDF, counts, years_exp, seniority).

Livrables : embeddings stockés + features table.

Phase 3 — Matching & Ranking (3–5 semaines)

- Baseline : cosine similarity sur embeddings + FAISS index.
- Avancé : fine-tuning BERT pour classification/ranking.
- Combinaison features structurés via un modèle de ranking (XGBoost / LightGBM).

Livrables : modèle de ranking, API de recherche, tests de performance.

Phase 4 — Débiaisation & Fairness (2–3 semaines)

- Mesures de biais (par genre, âge, origine si disponible).

- Techniques : rééchantillonnage, calibration des scores, post-processing équitable.

Livrables : rapport fairness + pipeline désensibilisation.

Phase 5 — Déploiement & Interface (2–4 semaines)

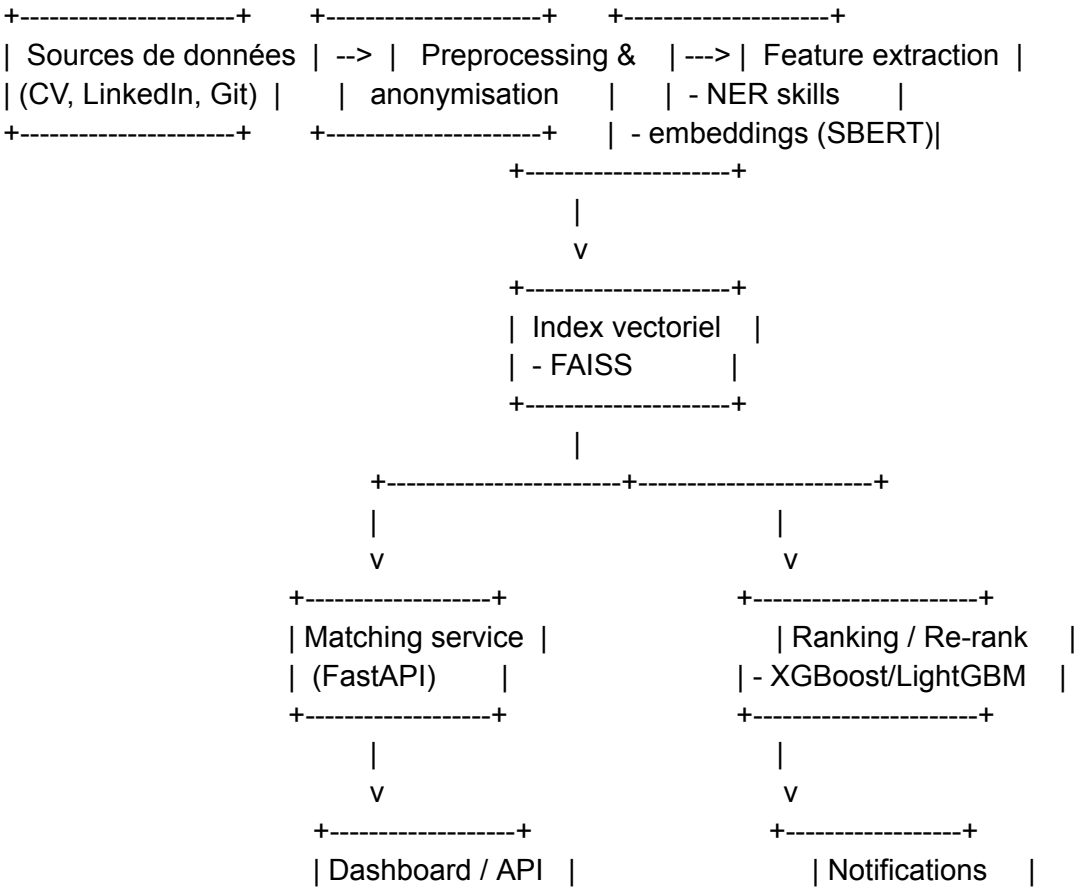
- Service de matching (FastAPI), indexation (FAISS), dashboard (React).
- Notifications proactives (emails/Slack) vers candidats et entreprises.

Livrables : démo fonctionnelle, documentation API, guide RH.

Phase 6 — Évaluation & itération (continu)

- Tests A/B pour valider recommandations.
- Feedback loop RH => réentraînement.

3) Architecture proposée (diagramme + composants)



| - Recherche RH | | - Emails/Slack |
+----------+ +----------+

Composants techniques

- Stockage embeddings : vector DB (FAISS / Milvus / Weaviate)
- Extraction skills : spaCy custom NER / rules / gazetteers
- Embeddings : Sentence-BERT (par exemple `all-MiniLM-L6-v2` ou plus grand)
- Search : FAISS + hnsw / IVF depending on scale
- Ranking : gradient boosting ou modèle neural supervisé
- API : FastAPI
- Dashboard : React + Tailwind (ou MUI)

4) Jeu de données — suggestions & schéma

Sources possibles

- CV anonymisés internes
- Profils LinkedIn (via API ou export)
- Résumés publics sur GitHub, Kaggle, papers (Google Scholar)
- Offres d'emploi publiques (pour matching)
- Datasets publics : `Kaggle résumés`, `Job Descriptions datasets`, `Glassdoor` dumps, `Open Skills` datasets

Remarque éthique : respecter RGPD/consentement ; anonymiser PII pour entraînement.

Schéma recommandé (CSV/Parquet)

- profile_id: string

- source: enum (cv, linkedin, github, internal)
- raw_text: string
- name: (anonymisé) string
- years_experience: int
- skills: list[string]
- positions: list[string]
- projects: list[{title, description, impact_metrics}]
- education: list[string]
- language: string
- created_at: date
- annotated_talent_score: float (optionnel pour entraînement supervisé)

Jeu de données synthétique (exemple 3 lignes)

profile_id,source,raw_text,years_experience,skills,positions,projects,language
p_001,cv,"Senior backend dev with 8 years...",8,"Python;Django;SQL","Senior Backend Engineer","[{...}]",fr
p_002,linkedin,"Data scientist, published 3 papers...",5,"Python;Pytorch;NLP","Data Scientist","[{...}]",en
p_003,github,"Open-source contributor: 20 repos...",3,"Go;Docker;K8s","DevOps Engineer","[{...}]",en

5) Code du modèle & pipeline (fichiers clefs)

> Le code ci-dessous est un squelette prêt à être exécuté localement. Il montre : prétraitement, extraction skills simple, embeddings SBERT, index FAISS, recherche basique et re-ranking par XGBoost.

5.1 `requirements.txt`

transformers
sentence-transformers
faiss-cpu
scikit-learn
xgboost

```
pandas
fastapi
uvicorn
spacy
python-multipart
```

5.2 `data_prep.py`

```
# data_prep.py
import pandas as pd
import re
from sentence_transformers import SentenceTransformer

MODEL_NAME = 'all-MiniLM-L6-v2'

def clean_text(text):
    text = re.sub(r"\s+", " ", text)
    return text.strip()

def load_csv(path):
    df = pd.read_csv(path)
    df['raw_text'] = df['raw_text'].fillna('')
    df['clean_text'] = df['raw_text'].apply(clean_text)
    return df

def compute_embeddings(df, model_name=MODEL_NAME):
    model = SentenceTransformer(model_name)
    texts = df['clean_text'].tolist()
    embeddings = model.encode(texts, show_progress_bar=True, convert_to_numpy=True)
    return embeddings

if __name__ == '__main__':
    df = load_csv('profiles.csv')
    emb = compute_embeddings(df)
    import numpy as np
    np.save('embeddings.npy', emb)
    df.to_parquet('profiles.parquet', index=False)
```

5.3 `index_search.py` (FAISS index)

```
# index_search.py
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer
import pandas as pd
```

```
def build_index(emb_path, dim, index_path='faiss.index'):
    embeddings = np.load(emb_path)
    index = faiss.IndexFlatIP(dim)  # inner product -> cosine if normalized
    faiss.normalize_L2(embeddings)
    index.add(embeddings)
    faiss.write_index(index, index_path)
    print('Index built, n =', index.ntotal)

def search(query, index_path='faiss.index', model_name='all-MiniLM-L6-v2', k=10):
    index = faiss.read_index(index_path)
    model = SentenceTransformer(model_name)
    q_emb = model.encode([query], convert_to_numpy=True)
    faiss.normalize_L2(q_emb)
    D, I = index.search(q_emb, k)
    return D[0], I[0]

if __name__ == '__main__':
    build_index('embeddings.npy', dim=384)
    d, ids = search('Experienced NLP engineer with pytorch experience', k=5)
    print(d, ids)
```

5.4 `ranking.py` (réentrainement léger XGBoost pour rerank)

```
# ranking.py
import xgboost as xgb
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

# Exemple: on suppose un jeu avec features déjà calculées
# features.csv contient: profile_id, sim_score, years_exp, publications_count, labelled_talent
(0/1)

def train_ranking(path='features.csv'):
    df = pd.read_csv(path)
    X = df[['sim_score','years_exp','publications_count']]
    y = df['labelled_talent']
    X_train, X_val, y_train, y_val = train_test_split(X,y,test_size=0.2,random_state=42)
    dtrain = xgb.DMatrix(X_train, label=y_train)
    dval = xgb.DMatrix(X_val, label=y_val)
    params = {'objective':'binary:logistic','eval_metric':'auc','max_depth':4}
    model = xgb.train(params, dtrain, num_boost_round=100, evals=[(dval,'val')],
early_stopping_rounds=10)
    model.save_model('xgb_rank.model')
    preds = model.predict(dval)
    print('AUC', roc_auc_score(y_val, preds))
```

```
if __name__ == '__main__':
    train_ranking()
```

5.5 `api_service.py` (FastAPI minimal)

```
# api_service.py
from fastapi import FastAPI
from pydantic import BaseModel
from index_search import search
import pandas as pd

app = FastAPI()
profiles = pd.read_parquet('profiles.parquet')

class Query(BaseModel):
    text: str
    k: int = 10

@app.post('/search')
async def search_profiles(q: Query):
    D, I = search(q.text, k=q.k)
    results = []
    for score, idx in zip(D, I):
        row = profiles.iloc[idx].to_dict()
        row['sim_score'] = float(score)
        results.append(row)
    return {'results': results}
```

6) Évaluation & metrics

Metrics proposées

- Precision@K (K = 5,10)
- Recall@K
- MAP (mean average precision)
- NDCG@K
- AUC (si task binaire)
- Temps de latence médian / p95 pour requêtes

Plan d'évaluation

1. Créer un jeu annoté : 1000 profils avec labels talent/non-talent (panel RH).
2. Calculer metrics baseline (cosine on embeddings).
3. Comparer modèles (baseline vs ranker XGBoost vs fine-tuned BERT).
4. Tests d'utilisabilité RH : taux d'acceptation des suggestions.

7) Fairness, privacy & conformité

- Anonymiser PII dans données d'entraînement.
- Expliquer décision (explainability) : fournir features qui ont mené au score (top-5 raisons).
- Audits de biais réguliers ; métriques par sous-groupes.
- Conserver consentements; possibilité d'opting-out.

8) Présentation PowerPoint — contenu des slides

Structure recommandée (12–15 slides)

1. Titre, auteur, date
2. Contexte & problématique
3. Objectifs
4. Questions de recherche / hypothèses
5. Architecture (diagramme)
6. Pipeline de données
7. Représentation & modèles (embeddings, FAISS, ranker)
8. Jeu de données & annotations

9. Métriques d'évaluation
10. Résultats (baseline vs modèle) — placeholder
11. Fairness & privacy
12. Démo & cas d'usage
13. Plan de déploiement
14. Roadmap & prochaines étapes
15. Q&A

Pour chaque slide : points clés + notes orateur (fournies dans le ppt si demandé).

9) Checklist pour rendre le projet reproductible

- Scripts pour nettoyer et anonymiser les données
- Notebook pour génération d'embeddings
- Script pour index FAISS et sauvegarde
- API + dockerfile
- Tests unitaires pour fonctions clefs
- Jeu de données annoté (train/val/test)
- Documentation d'évaluation et fairness

10) Propositions d'extensions avancées

- Multi-modal (CVs + code snippets + vidéos d'entretien)
- Graph-based: construire un knowledge graph compétences–projets–entreprises
- Active learning pour annotation RH
- Système de négociation automatique entre entreprises et talents (marketplace)

Étapes du travail:

✅ Étape 1 — Définir clairement les sources de données

Comme ton système doit *découvrir* les talents même si eux ne postulent pas, tu dois utiliser des **sources où les gens publient déjà des traces de leurs compétences**, même sans intention de chercher un job.

Voici les meilleures sources **réalistes + adaptées au NLP** 👇

1) Profils professionnels publics

Ce sont les données les plus importantes, car les talents y écrivent déjà leurs expériences.

🔹 LinkedIn (public profiles)

- Expériences
- Compétences
- Certifications
- Recommandations
- Bio / résumé

✔️ C'est la **source principale** dans 90% des systèmes de talent scouting.

⚠️ Mais pour un projet d'école : on prend *des profils fictifs ou anonymisés*.

2) CV disponibles publiquement

Exemples :

- CV postés sur GitHub
- CV mis en ligne sur des portfolios
- CV partagés dans des projets open-source

✔️ Faciles à collecter, utilisables dans un projet NLP, données souvent structurées.

3) Plateformes techniques (pour détecter les vrais talents tech)

🔹 GitHub

- Qualité du code
- Contributions
- Projets
- Langages utilisés

→ Très puissant pour détecter les **talents techniques**.

🔹 Kaggle

- Classements
- Compétences data
- Projets
- Notebooks

→ Idéal pour repérer des Data Scientists.

🟦 4) Publications & articles professionnels

- Google Scholar
- Medium (articles tech)
- ArXiv
- Blog techniques

✔️ Détecte les talents **scientifiques / chercheurs / experts IA**.

🟦 5) Données internes (si ton système est pour une entreprise)

- Évaluations internes
- Feedback managers
- Projets réalisés
- Emails professionnels (résumés, pas du contenu sensible)
- Rapports de performance

⚠️ Ces données sont puissantes mais sensibles (RGPD). Pour ton projet d'école, **tu n'en utiliseras pas**.

Lancement du travail:

1/Scraping:

On va adapter le script pour scraper plusieurs utilisateurs GitHub automatiquement et générer un CSV global prêt à être utilisé pour NLP:

Script complet pour plusieurs utilisateurs:

```
import requests

import base64

import time

import pandas as pd


# Optionnel : ajouter un token GitHub pour éviter les limitations

GITHUB_TOKEN = None  # exemple: "ghp_xxxxx"

HEADERS = {"Authorization": f"token {GITHUB_TOKEN}"} if GITHUB_TOKEN else {}


# Liste des utilisateurs GitHub à scraper

usernames = ["torvalds", "mojombo", "pjhyett", "defunkt"]  # tu peux ajouter plus de pseudos


# Liste pour stocker tous les projets

all_projects = []
```

```
def get_user_repos(username):
    url = f"https://api.github.com/users/{username}/repos"
    r = requests.get(url, headers=HEADERS)
    if r.status_code != 200:
        print(f"Erreur pour {username} :", r.text)
        return []
    return r.json()


def get_readme(owner, repo):
    url =
f"https://api.github.com/repos/{owner}/{repo}/readme"
    r = requests.get(url, headers=HEADERS)
    if r.status_code != 200:
        return ""
    data = r.json()
    if "content" in data:
        try:
```

```
            return
base64.b64decode(data["content"]).decode("utf-8",
errors="ignore")
        except:
            return ""
    return ""


def scrape_github_user(username):
    repos = get_user_repos(username)
    user_projects = []
    for repo in repos:
        name = repo.get("name", "")
        description = repo.get("description", "") or ""
        language = repo.get("language", "") or ""
        stars = repo.get("stargazers_count", 0)
        readme = get_readme(username, name)

        user_projects.append({
            "username": username,
            "project_name": name,
```

```
            "language": language,
            "stars": stars,
            "description": description,
            "readme": readme
        })

        time.sleep(1)  # éviter le rate limit
    return user_projects

# Scraper tous les utilisateurs
for user in usernames:
    print(f"Scraping {user}...")
    projects = scrape_github_user(user)
    all_projects.extend(projects)

# Convertir en DataFrame
df = pd.DataFrame(all_projects)

# Sauvegarder dans un CSV prêt pour NLP
```

df.to_csv("github_profiles_dataset.csv", index=False, encoding="utf-8")

print("✅ CSV généré : github_profiles_dataset.csv")

- **Ce que fait ce script**

1. Prend **une liste de pseudos GitHub** (`usernames`)
2. Pour chaque utilisateur :
    - Récupère tous ses projets publics
    - Récupère description, langage, stars et README
3. Stocke tout **dans une seule table (DataFrame)**
4. Sauvegarde dans **un CSV global**, prêt pour NLP

Prochaines étapes NLP

1. Nettoyage des textes (README + description)
2. Extraction de compétences / mots-clés

3. Génération d’embeddings pour matching talent ↔ offre

4. Calcul du score de talent / classement