

Classification Multi-Label de Tags Codeforces

Présentation Technique - Entretien Data Scientist

Zakaria El Kazdam

Formulation du Problème

Contexte

Dataset : 4982 problèmes algorithmiques Codeforces

- **Format** : Chaque problème est un fichier JSON contenant :

- Description du problème (``prob_desc_description``)

- Code source de référence (``source_code``)

- Tags associés (``tags``)

- Autres métadonnées (input_spec, output_spec, etc.)

Objectif

- **Tâche** : Classification multi-label

- **Input** : Description et/ou code source d'un problème

- **Output** : Ensemble de tags prédits (chaque problème peut avoir plusieurs tags)

- **Tags cibles** : 8 focus tags : ``math``, ``graphs``, ``strings``, ``number theory``, ``trees``, ``geometry``, ``games``, ``probabilities``

Formulation du Problème

Défis

- Déséquilibre des classes : ``math`` 28% vs ``probabilities`` 1.8%
- Multi-label : Chaque problème peut avoir 1 à plusieurs tags simultanément
- Dataset réduit : Après filtrage aux 8 focus tags → ~2678 exemples (53% du dataset original)
- Petit dataset : Limite les approches nécessitant beaucoup de données (ex: Word2Vec, transformers)

----- Basic Statistics -----

Average number of tags per problem: 2.80

Median number of tags: 3

Min tags: 0

Max tags: 11

Problems with 0 tags: 41

Problems with 1 tag: 857

Problems with 2 tags: 1455

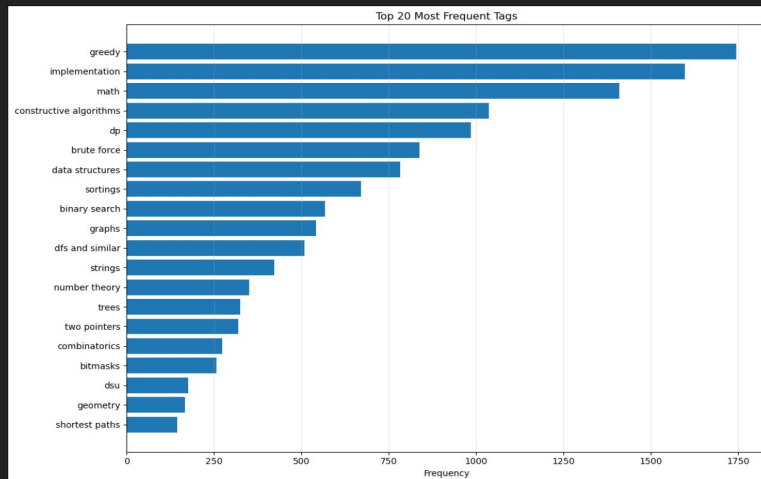
Problems with 3+ tags: 2629

----- Focus Tags Statistics -----

tag	count	percentage
math	1409	28.281815
graphs	542	10.879165
strings	422	8.470494
number theory	350	7.025291
trees	324	6.503412
geometry	166	3.331995
games	105	2.107587
probabilities	92	1.846648

Problems with at least one focus tag: 2678 (53.8%)

Problems with NO focus tags: 2304 (46.2%)



Extraction de Features - Pourquoi Description et Code ?

Features disponibles dans le dataset

Chaque problème JSON contient plusieurs champs :

- ``prob_desc_description`` : Description textuelle du problème (naturel)
- ``source_code`` : Code source de la solution de référence (code)
- ``input_spec`` : Spécification de l'input (format, contraintes)
- ``output_spec`` : Spécification de l'output attendu
- ``tags`` : Tags associés (labels)

Choix : Description + Code

Pourquoi Description ?

- Contient l'énoncé du problème en langage naturel
- Décrit la logique, les contraintes, le contexte
- **Exemple** : "Given an array, find the maximum sum" → tag ``math`` ou ``arrays``
- **Information riche** : Capture le "quoi" du problème

Extraction de Features - Pourquoi Description et Code ?

Pourquoi Code ?

- Montre la solution concrète, l'algorithme utilisé
- **Exemple** : Code avec DFS/BFS → tag ``graphs``, ``trees``
- **Information complémentaire** : Capture le "comment" résoudre
- **Patterns algorithmiques** : Structures de données, algorithmes classiques

Pourquoi PAS input_spec / output_spec ?

- Données manquantes
- Redondant avec description/code
- **Décision** : Focus sur description + code pour maximiser l'information textuelle

Flexibilité

- Choix via CLI : `--features description`` ou `--features code`` ou `--features description,code``
- Chaque feature vectorisée ****indépendamment**** puis concaténée

----- Feature Availability Analysis -----

Total files analyzed: 4982

--- DESCRIPTION (prob_desc_description) ---

Present in JSON: 4982/4982 (100.0%)
Missing from JSON: 0/4982 (0.0%)
Non-empty values: 4982/4982 (100.0%)
Empty/None values: 0/4982 (0.0%)

--- INPUT_SPEC (prob_desc_input_spec) ---

Present in JSON: 4982/4982 (100.0%)
Missing from JSON: 0/4982 (0.0%)
Non-empty values: 4949/4982 (99.3%)
Empty/None values: 33/4982 (0.7%)

--- OUTPUT_SPEC (prob_desc_output_spec) ---

Present in JSON: 4982/4982 (100.0%)
Missing from JSON: 0/4982 (0.0%)
Non-empty values: 4897/4982 (98.3%)
Empty/None values: 85/4982 (1.7%)

--- SOURCE_CODE (source_code) ---

Present in JSON: 4982/4982 (100.0%)
Missing from JSON: 0/4982 (0.0%)
Non-empty values: 4982/4982 (100.0%)
Empty/None values: 0/4982 (0.0%)

Preprocessing

Preprocessing de la Description

- Suppression des délimiteurs LaTeX : ``$$$`` (artefacts du parsing)
- Normalisation des espaces : Multiples espaces → un seul espace
- Raison : LaTeX pollue le texte, mais on garde le contenu mathématique

Preprocessing du Code

- Normalisation des espaces uniquement : Multiples espaces → un seul
- Pas de tokenisation spéciale : On garde la structure du code
- Raison : Le code contient des patterns importants (noms de fonctions, structures)

Filtrage des Tags

- Focus sur 8 tags : ``math``, ``graphs``, ``strings``, ``number theory``, ``trees``, ``geometry``, ``games``, ``probabilities``
- Suppression des exemples sans tags : Après filtrage, on garde seulement les problèmes ayant au moins un des 8 tags
- Résultat : Dataset réduit de 4982 → ~2678 exemples (53%)

Méthodes d'Embedding - TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency)

Principe : Mesure l'importance d'un mot dans un document par rapport à la collection

Paramètres choisis

- ``ngram_range=(1,2)`` : Unigrammes + bigrammes
- **Raison** : Capture les expressions importantes ("binary search", "graph traversal")
- ``min_df=2`` : Ignore les mots apparaissant dans < 2 documents
- **Raison** : Élimine les typos/artefacts, garde les mots significatifs
- ``max_df=0.9`` : Ignore les mots dans > 90% des documents
- **Raison** : Élimine les mots trop fréquents (stopwords)
- ``sublinear_tf=True`` : Log de la fréquence au lieu de fréquence brute ($tf = 1 + \log(freq)$)
- **Raison** : Réduit l'impact des mots très fréquents dans un document

Avantages

- **Rapide** : ~1-2 secondes pour 2000 exemples
- **Efficace sur petits datasets** : Pas besoin de beaucoup de données
- **Capture les patterns locaux** : Bigrammes importants pour les tags
- **Matrices sparse** : Efficace en mémoire

Résultat

- Matrice sparse de shape `(n_examples, n_features)`` où ``n_features`` = vocabulaire unigrammes + bigrammes

Méthodes d'Embedding - Word2Vec

Principe : Apprend des représentations vectorielles denses pour les mots en prédisant le contexte

Architecture

- **Skip-gram** : Prédit les mots du contexte à partir d'un mot central
- **Entraînement** : Sur le dataset Codeforces (pas de modèle pré-entraîné)
- **Tokenisation** : Split simple sur espaces (limitation actuelle)

Paramètres choisis

- ``vector_size=100`` : Dimension des embeddings (compromis taille/performance)
- ``min_count=2`` : Ignore les mots apparaissant < 2 fois
- ``epochs=10`` : Nombre d'itérations sur le corpus
- ``window=5`` : Taille du contexte (5 mots avant/après)
- ``sg=1`` : Skip-gram

Transformation en vecteur de document

- Pour chaque texte : Moyenne des vecteurs des mots présents dans le vocabulaire
- Si aucun mot connu : Vecteur de zéros

Avantages/Inconvénients

- **Représentations sémantiques** : Capture les similarités entre mots
- **Nécessite beaucoup de données** : Dataset trop petit (2678 exemples)
- **Moins performant**
- **Plus lent** : ~10-15 secondes pour 2000 exemples

Résultat

- Matrice dense de shape ``(n_examples, 100)`` (vector_size)

CodeBERT

Principe : Transformer pré-entraîné sur code (Microsoft, basé sur RoBERTa)

Architecture

- **Modèle** : ``microsoft/codebert-base`` (~125M paramètres)
- **Tokenisation** : BPE (Byte-Pair Encoding) spécialisé pour le code
- **Embedding** : Token [CLS] de la dernière couche (768 dimensions)

Paramètres

- **max_length=512** : Tronque si texte plus long (perte d'info possible)
- **padding=True** : Pad les séquences courtes dans un batch

Utilisation

- **Optimisé pour le code** : Pas recommandé pour descriptions (warning affiché)
- **Chargement** : Télécharge ~499MB au premier usage
- **Pas de cache** : Recharge le modèle à chaque appel (simplicité)

Avantages/Inconvénients

- **Haute qualité** : Embeddings contextuels riches
- **Optimisé pour code** : Comprend les structures de code
- **Très lent sur CPU** : ~5-10 minutes pour 2000 exemples
- **GPU recommandé** : Pratique uniquement avec GPU

Résultat

- Matrice dense de shape ``(n_examples, 768)``

Architecture du Modèle de Classification

Architecture du Classifieur

- **Stratégie** : ``OneVsRestClassifier`` (One-vs-Rest)
 - **Principe** : Un classifieur binaire par tag (tag présent vs absent)
 - **Raison** : Standard pour classification multi-label
 - **Avantage** : Simple, efficace, interprétable
-
- **Classifieur de base** : ``LogisticRegression``
 - `class_weight="balanced"` : Poids inversement proportionnel à la fréquence de classe
 - **Raison** : Gère le déséquilibre (math: 28% vs probabilités: 1.8%)

Vectorisation Multi-Features

1. **Vectorisation indépendante** : Description et code vectorisés séparément
2. **Concaténation horizontale** : ``[embedding_desc | embedding_code]``

Pipeline complet

Description → Embedding → [Desc_emb]

Code → Embedding → [Code_emb]

[Desc_emb | Code_emb]

OneVsRest(LogisticRegression)

[prob_tag1, prob_tag2, ..., prob_tag8]

Threshold (0.5) → Tags prédits

Métriques d'Évaluation

Métriques par Classe (Classification Report)

Pour chaque tag, on calcule :

- **Precision, Recall, F1-score**
- **Support** : Nombre d'exemples réels avec ce tag

Agrégations

- **Micro-avg** : Calcule les métriques globales (tous les tags confondus)
- **Macro-avg** : Moyenne des métriques par classe (non pondérée)
- **Weighted-avg** : Moyenne pondérée par le support de chaque classe

Jaccard Similarity

- **Formule** : $|A \cap B| / |A \cup B|$ où A = tags réels, B = tags prédits
- **Interprétation** : Similarité entre l'ensemble de tags réel et prédit
- **Range** : [0, 1] où 1 = parfait match
- **Average** : ``average='samples'`` - Moyenne sur tous les exemples

Hamming Loss

- **Formule** :
- **Interprétation** : Fraction de labels incorrectement prédits
- **Range** : [0, 1] où 0 = parfait, 1 = toutes les prédictions fausses
- **Avantage** : Pénalise à la fois les faux positifs ET les faux négatifs

$$\text{Hamming Loss} = \frac{\text{Number of incorrect label decisions}}{\text{Number of labels}}$$

Traçabilité - Rapports et Modèles Sauvegardés

Rapports JSON Automatiques

Chaque entraînement génère un rapport JSON complet avec :

Informations de Configuration

- Features utilisées (`description`, `code`, ou les deux)
- Types d'embedding pour chaque feature (`tfidf`, `word2vec`, `codebert`)
- Paramètres d'embedding (Word2Vec, TF-IDF)
- Paramètres du classifieur (solver, class_weight, etc.)
- Split train/val/test (proportions, random_state)
- Focus tags activé ou non

Résultats et Métriques

- Classification report complet (precision, recall, F1 par tag)
- Métriques multi-label (Jaccard similarity, Hamming loss)
- Agrégations (micro-avg, macro-avg, weighted-avg)

Traçabilité

- IDs du test set : Liste complète des problem_id dans le test set
- Timestamp : Date et heure de l'entraînement
- Nom du rapport : Format

`report_{embedding}_{features}_{classfier}_{timestamp}.json`

- Exemple :

`report_tfidf_description_tfidf_code_LogisticRegression_01_19_16_10.json`

ts > {} report_code_tfidf_description_tfidf_code_description_LogisticRegression_01_19_21_13.json >

```
{
  "features": [
    "description",
    "code"
  ],
  "embedding_types": {
    "description": "tfidf",
    "code": "tfidf"
  },
  "embedding_params": {
    "description": {
      "ngram_range": [
        1,
        2
      ],
      "min_df": 2,
      "max_df": 0.9,
      "sublinear_tf": true
    },
    "code": {
      "ngram_range": [
        1,
        2
      ],
      "min_df": 2,
      "max_df": 0.9,
      "sublinear_tf": true
    }
  },
  "classifier": "LogisticRegression",
  "classifier_params": {
    "solver": "liblinear",
    "class_weight": "balanced",
    "max_iter": 250,
    "strategy": "OneVsRest"
  },
  "train_size": 0.7,
  "val_size": 0.15,
  "test_size": 0.15000000000000005,
  "random_state": 42,
  "focus_tags_only": true,
  "validation_metrics": {
    "games": {
      "precision": 0.8333333333333334,
      "recall": 0.6666666666666666,
      "f1-score": 0.7407407407407407,
      "support": 15.0
    }
  },
}
```

Traçabilité - Rapports et Modèles Sauvegardés

Modèles Sauvegardés

- **Format** : ``.joblib`` (sérialisation Python)
- **Contenu** : Vectorizers, classifieur, MultiLabelBinarizer, embedding types, features
- **Chemin** : Spécifié par l'utilisateur via ``--model-path``
- **Usage** : Chargement direct pour prédictions sans réentraînement

Avantages

- **Reproductibilité** : Toutes les configurations sauvegardées
- **Comparaison** : Facile de comparer différents modèles
- **Traçabilité** : Test set IDs préservés pour éviter data leakage
- **Historique** : Timestamp permet de suivre l'évolution

```
▼ models
  ≡ tfidf_logreg_CodeOnly.joblib
  ≡ tfidf_logreg_DescCode.joblib
  ≡ tfidf_logreg_DescOnly.joblib
▼ notebooks
  📄 01_data_analysis.ipynb
▼ reports
  {} report_code_tfidf_description_tfidf_code_description_LogisticRegression_01_19_21_13.json
  {} report_code_tfidf_description_tfidf_code_description_LogisticRegression_01_20_07_27.json
  {} report_tfidf_code_LogisticRegression_01_19_16_10.json
  {} report_tfidf_description_LogisticRegression_01_19_16_07.json
  {} report_tfidf_description_LogisticRegression_01_19_16_08.json
```

Interface CLI et Flexibilité

Interface en Ligne de Commande

Le projet expose une CLI simple et intuitive via ``argparse`` :

Commande ``train`` : Entraîner un nouveau modèle

- ``--data-dir`` : Répertoire contenant les fichiers JSON
- ``--model-path`` : Où sauvegarder le modèle entraîné
- ``--features`` : Choix des features (``description``, ``code``, ou ``description,code``)
- ``--embedding-desc`` / ``--embedding-code`` : Choix indépendant de l'embedding pour
- ``--word2vec-vector-size``, ``--word2vec-min-count``, ``--word2vec-epochs`` : Paramètres Word2Vec
- ``--train-size``, ``--val-size``, ``--random-state`` : Paramètres de split
- ``--focus-tags-only`` : Filtrer aux 8 focus tags

Flexibilité du Système

1. Choix des Features

- Description seule, code seul, ou les deux
- Permet d'évaluer la contribution de chaque feature

2. Choix des Embeddings

- Choix indépendant : peut mixer (ex: TF-IDF pour description, CodeBERT pour code) / Warning si CodeBERT utilisé pour description (non recommandé)

3. Paramètres Modifiables

- Tous les hyperparamètres accessibles via CLI
- Pas besoin de modifier le code pour changer les paramètres
- Facilite l'expérimentation et la comparaison

Commande ``predict`` : Prédire des tags

- ``--model-path`` : Modèle à utiliser
- ``--json-file`` : Fichier unique à prédire
- ``--data-dir`` : Répertoire de fichiers à prédire (batch)
- ``--threshold`` : Seuil de probabilité pour les prédictions

Exemples d'Utilisation

Exemple 1 : TF-IDF avec description + code

```
```bash
python -m code_classifier.cli train \
--data-dir ./data \
--model-path models/tfidf_model.joblib \
--features description,code \
--embedding-desc tfidf \
--embedding-code tfidf \
--focus-tags-only
```
```

Exemple 2 : Word2Vec pour description

```
```bash
python -m code_classifier.cli train \
--data-dir ./data \
--model-path models/w2v_model.joblib \
--features description \
--embedding-desc word2vec \
--word2vec-vector-size 150 \
--focus-tags-only
```
```

Exemple 3 : CodeBERT pour code, TF-IDF pour description

```
```bash
python -m code_classifier.cli train \
--data-dir ./data \
--model-path models/codebert_model.joblib \
--features description,code \
--embedding-desc tfidf \
--embedding-code codebert \
--focus-tags-only
```
```

```
(.venv) zakaria@zakaria-IdeaPad-Slim-3-15IRH8:~/Downloads/ILLUIN Technology/code_classification_dataset$ python -m
tfidf_model.joblib --features description,code --embedding-desc tfidf --embedding-code tfidf --focus-tags-only
Using features: ['description', 'code']
Loaded 4982 examples
After filtering to focus tags: 2678 examples
Description embedding: tfidf
Code embedding: tfidf
Model saved to: models/tfidf_model.joblib
Report saved to: reports/report_code_tfidf_description_tfidf_code_description_LogisticRegression_01_20_07_27.json
Test set contains 403 examples
```

```
=== Validation Metrics ===
{
  "games": {
    "precision": 0.8333333333333334,
    "recall": 0.6666666666666666,
    "f1-score": 0.7407407407407407,
    "support": 15.0
  },
  "geometry": {
    "precision": 0.6956521739130435,
    "recall": 0.7619047619047619,
    "f1-score": 0.7272727272727273,

```

Résultats et Performances

TF-IDF (Description + Code)

- Temps d'entraînement : ~4-5 secondes (2678 exemples après filtrage)
- Temps de prédiction : < 1 seconde pour 1000 exemples
- Performance :
 - F1-score macro-avg : ~ 0.71
 - F1-score weighted-avg : ~ 0.75
 - Jaccard similarity : ~ 0.67
 - Hamming loss : ~0.07
- Meilleurs tags : `math`, `strings`, `graphs` (classes majoritaires)
- Tags difficiles : `probabilities`, `games` (classes minoritaires)

Décision Finale

- TF-IDF recommandé : Meilleur compromis vitesse/performance
- Description + Code : Meilleure performance que description seule
- Word2Vec : Utile pour comparaison mais moins adapté au petit dataset

Word2Vec (Description uniquement)

- Temps d'entraînement : ~10-15 secondes (entraînement Word2Vec inclus)
- Performance :
 - F1-score macro-avg : ~0.57 (moins performant)
 - Raison : Dataset trop petit pour Word2Vec efficace
- Conclusion : Word2Vec nécessite plus de données pour être compétitif

CodeBERT (Code uniquement)

- Temps d'entraînement : ~5-10 minutes sur CPU (2678 exemples)
- Performance : Non testé complètement (trop lent pour itérations rapides)
- Recommandation : Utiliser uniquement avec GPU pour être pratique

Points Forts du Projet

1. Architecture Propre et Modulaire

- Séparation des responsabilités : ``data.py``, ``preprocessing.py``, ``model.py``, ``cli.py``
- Code extensible : Ajout facile de nouvelles méthodes d'embedding (ex: CodeBERT)
- Interface CLI intuitive : Arguments clairs, flexibilité des features

2. Bonnes Pratiques Data Science

- Train/val/test split : Évite le data leakage, évaluation rigoureuse
- Sauvegarde complète : Configurations, métriques, IDs test set dans rapports JSON
- Métriques appropriées : Classification report + Jaccard + Hamming loss

3. Approche Méthodique

- EDA préalable : Notebook d'analyse des données (``01_data_analysis.ipynb``)
- Comparaison de méthodes : TF-IDF vs Word2Vec vs CodeBERT
- Documentation complète : README détaillé, docstrings, commentaires
- **Reproductibilité** : ``random_state=42``, sauvegarde des configurations

Limitations Actuelles

1. Dataset Réduit

- 2678 exemples après filtrage (53% du dataset original)
- Limite les approches nécessitant beaucoup de données (Word2Vec, transformers)

2. Word2Vec Sous-Performant

- Raison : Dataset trop petit pour apprendre de bons embeddings

3. Tokenisation Simple

- Split sur espaces uniquement
- Pas de gestion spéciale pour le code (opérateurs, ponctuation)

4. CodeBERT Non Optimisé

- Trop lent sur CPU pour itérations rapides

5. Hyperparamètres Non Optimisés

- Paramètres TF-IDF et classifieur choisis empiriquement
- Pas de grid search ou random search

Améliorations Possibles

1. Embeddings

- `FastText` : Gère mieux les mots rares (subword information)
- `CodeBERT optimisé` : Cache du modèle, utilisation GPU
- `Embeddings pré-entraînés` : Utiliser Word2Vec/FastText pré-entraînés sur code

2. Preprocessing

- `Tokenisation sophistiquée` : tokenizers spécialisés pour code
- `Normalisation code` : Standardiser les noms de variables (ex: `var1`, `var2`)

3. Modèles de Classification

- `SVM` : Peut mieux gérer les non-linéarités
- `Random Forest` : Feature importance, robustesse
- `Neural Networks` : Pour datasets plus grands

4. Features Additionnelles

- ``input_spec`` et ``output_spec`` : Peuvent contenir des patterns utiles
- `Métadonnées` : Difficulté, nombre de solutions, etc.

5. Optimisation Hyperparamètres

- `Grid search` : TF-IDF (ngram_range, min_df, max_df)
- `Random search` : Classifieur (C, penalty, solver)

6. Analyse des Erreurs

- `Confusion matrix` : Quels tags sont confondus ?
- `Exemples mal classés` : Analyser les patterns d'erreur
- `Feature importance` : Quelles features contribuent le plus ?