NLP Resume Classification

A Comparative Analysis of Classical Machine Learning and Deep Transformer Models for Text Classification

By: Shesadree Priyadarshani, Nitishwar Vasanthakumar, Sibi Seenivasan
Term Project for EGN 5442

1. Abstract

Our project builds an automated system that scores resume-job description pairs into three classes: No Fit, Potential Fit, and Good Fit to support high-volume HR screening. We compared a feature-engineered classical pipeline (Logistic Regression, Linear SVC, XGBoost) against fine-tuned transformer models (DistilBERT/BERT) trained on a curated version of the HuggingFace resume-jd-match dataset. The best overall model is a tuned XGBoost classifier on TF-IDF and similarity features, achieving around 72–73% test accuracy, a weighted F1 ≈ 0.72, and ROC-AUC ≈ 0.87 on the held-out set, outperforming the transformer baselines in this setup.[1][2][3]

2. Problem and dataset

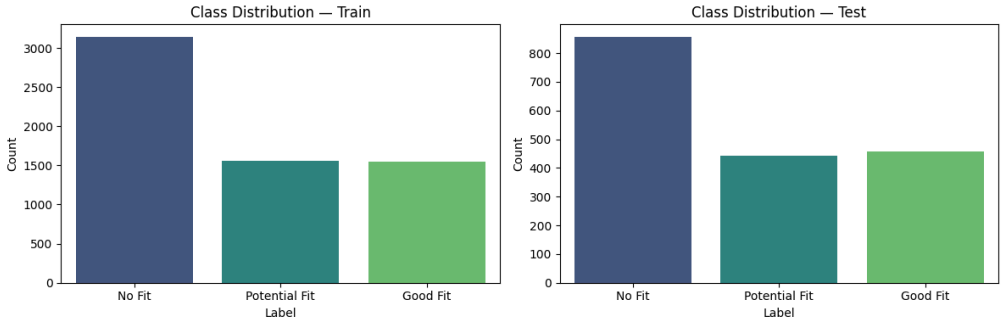
Modern HR teams receive thousands of resumes per opening, making manual matching to job descriptions slow, inconsistent, and vulnerable to bias toward easily recognized profiles. The goal is to learn a multi-class classifier that predicts candidate suitability purely from textual signals in the JD and resume, enabling consistent triage into No Fit, Potential Fit, and Good Fit buckets.[2][1]

For our project, we used the public HuggingFace dataset `facehugger/apoorv-resume-jd-match`, which provides 6,241 training and 1,759 test pairs of JD text, resume text, and a human-assigned label. Labels are highly imbalanced, with “No Fit” roughly twice as frequent as “Good Fit” and “Potential Fit,” so naive accuracy would over-reward predicting the majority class.[2]

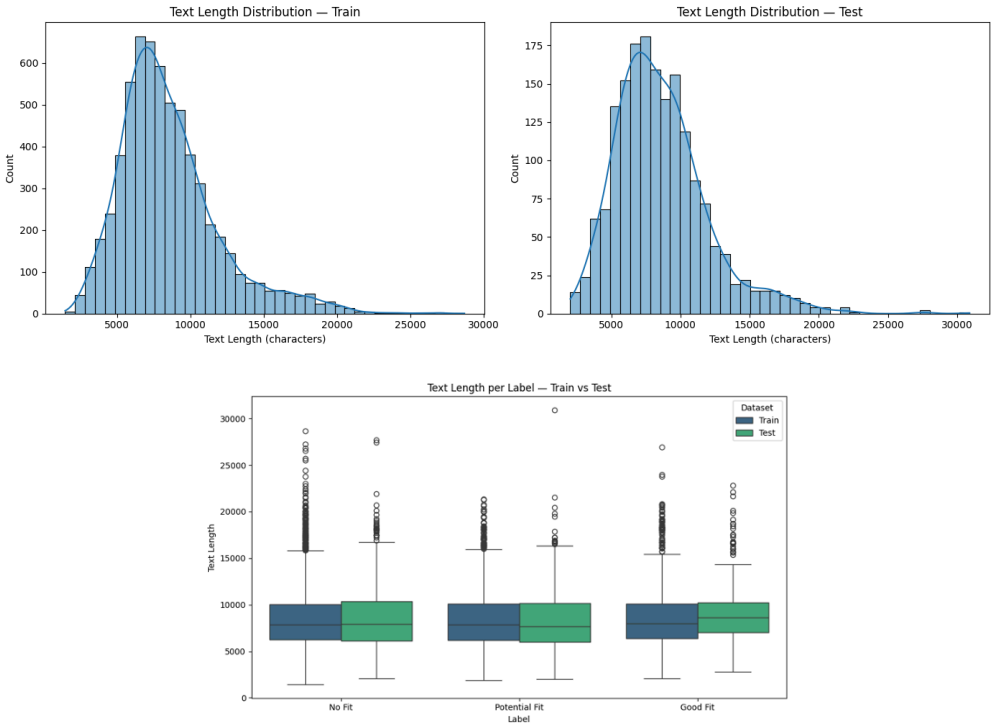
3. Data acquisition and exploratory analysis

The raw dataset is loaded directly from HuggingFace and converted into Pandas DataFrames for train (6,241 samples) and test (1,759 samples). Three labels are present: Good Fit, Potential Fit, and No Fit, which are later mapped to integer IDs for modeling.[2]

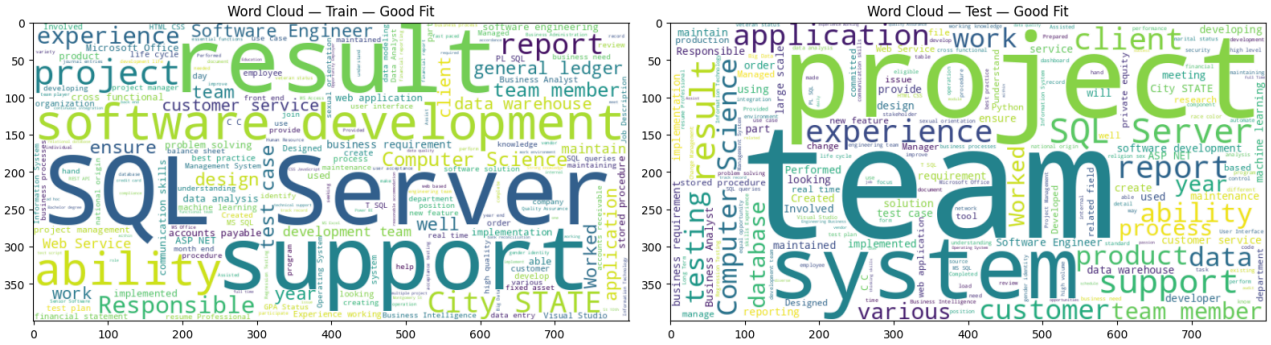
Class balance plots show “No Fit” as the dominant class in both train and test, followed by roughly equal counts of Good Fit and Potential Fit, confirming a 2:1 imbalance. This implies that a model can reach 60–70% accuracy by over-predicting “No Fit,” motivating the use of weighted F1 and per-class diagnostics instead of raw accuracy.[2]



Text-length histograms of the cleaned combined text indicate similar distributions between train and test, and boxplots by label and split show consistent medians and spreads, supporting good dataset stability.



Word-frequency and word-cloud analysis reveals that Good Fit cases emphasize domain-specific skills (e.g., “computer science”, “application”, “project”, “test case”), whereas No Fit profiles over-index on generic business terms like “support”, “experience”, and “team” without strong technical anchors.[2]



4. Data preprocessing and leakage prevention

The raw text for every example begins with a fixed instruction-tuning prefix of the form “For the given job description …”, confirming that the dataset was originally curated for LLM instruction tasks rather than classification. We removed this prefix via regex and string cleaning so that models can spend capability on the JD and resume content instead of memorizing a constant template that carries no discriminative signal.[2]

A second, more severe leakage issue is that the literal label phrases “good fit”, “no fit”, and “potential fit” appear inside many input texts. Before cleaning, every training row contains at least one label word, meaning any model could trivially “cheat” by reading the label from the prompt rather than learning semantic compatibility. Regex patterns are used to strip these label terms and related artifacts (e.g., “fit score”, “label ?”), and a post-check confirms zero remaining label words in the cleaned corpus, eliminating direct target leakage.[2]

Labels are then encoded with a shared `LabelEncoder` applied consistently to train and test, mapping string labels to integer IDs (e.g., Good Fit → 0, No Fit → 1, Potential Fit → 2 in the working notebooks). The cleaned text is further split into two features `jd_cleaned` and `resume_cleaned` by identifying the “the resume” delimiter, and several noisy sections (contact info, URLs, recruiter signatures, boilerplate EEO statements, “about us” fluff paragraphs) are removed from both JD and resume to reduce noise.[2]

5. Methodology I – classical machine learning

Because linear models and tree-based learners cannot operate directly on raw strings, the classical pipeline converts each JD and resume into sparse numerical feature vectors. Two separate TF-IDF vectorizers (unigrams and bigrams, English stop words, 12,000 features each) are fit on `jd_cleaned` and `resume_cleaned`, then concatenated.[1]

Additional engineered features aim to encode explicit JD–resume alignment. First, cosine similarity is computed between the JD and resume TF-IDF vectors for each pair, providing a dense single feature that measures global lexical overlap. Second, a lightweight skill-matching mechanism extracts high-TF-IDF JD keywords and counts both raw overlap and overlap ratio in the resume, yielding numeric features `skill_match_count` and `skill_ratio`. These numeric features are standardized (without centering, to preserve sparsity) and stacked with the TF-IDF matrices into a unified sparse design matrix.[1]

We then trained 3 machine learning models: Logistic Regression, Linear SVC, and XGBoost. Initially we used a train/validation/test split with shapes around 5,616/625/1,759 and simple evaluation loops; later, the data is re-split into approximately 5,600 train, 1,200 validation, and 1,200 test instances to support cross-validated hyperparameter tuning for XGBoost via `RandomizedSearchCV`.[1]

6. Methodology II – transformer models

The advanced pipeline replaces bag-of-words features with contextual embeddings, learning directly from raw token sequences of the concatenated JD and resume. Cleaned `jd_cleaned` and `resume_cleaned` text are tokenized with a DistilBERT/BERT tokenizer to a maximum of 512 tokens, with truncation applied to handle long resumes and job descriptions. The combined sequence is represented by pre-computed `input_ids` and `attention_mask` arrays that are stored and reused for training.[3][2]

The primary architectures explored are DistilBERT and BERT-base sequence-classification heads, initialized from pre-trained checkpoints and fine-tuned with a 3-way classifier head. The JD and resume are effectively modeled together by tokenizing them into a single sequence with segment structure preserved in the text, allowing the transformer self-attention layers to capture cross-document relationships implicitly; the notebooks operate on these concatenated representations via the standard `[CLS]` classification token.[3]

Class imbalance is addressed via weighted cross-entropy: class weights inversely proportional to class frequencies are passed to the loss function so that misclassifying Good Fit and Potential Fit incurs a larger penalty than misclassifying No Fit. Training uses GPU acceleration (NVIDIA L4), HuggingFace `Trainer` utilities, and standard validation-set early stopping, with evaluation focused on weighted F1 and ROC-AUC rather than accuracy alone.[3]

After rebuilding the dataset into a clean, consistent train, val & test split, we observed significant improvements in both accuracy and AUC, indicating better generalization and reduced noise. For the Cross Attention dual encoder model, addition of retokenized JD and Resume separately so that each encoder receives its own sequence, this enables explicit cross attention between JD tokens and Resume tokens. This architecture further explores deeper semantic alignment beyond standard concatenation and provides an additional path for performance improvement.

7. Results and comparative analysis

7.1 XGBoost model

For the classical models on the initial split, training metrics show progressively stronger fitting from Logistic Regression to SVM to XGBoost, with XGBoost nearly saturating the training set (F1 ≈ 0.999), indicating high capacity. On the original held-out test set (1,759 examples), baseline test performance is roughly: Logistic Regression accuracy ≈ 0.49 (weighted F1 ≈ 0.46), SVM accuracy ≈ 0.47 (F1 ≈ 0.46), and XGBoost accuracy ≈ 0.53 (weighted F1 ≈ 0.51).[1]

After rebuilding the splits and performing targeted hyperparameter search for XGBoost with a pipeline (standard scaling + `XGBClassifier`), the tuned XGBoost model achieves training accuracy ≈ 0.99 and weighted F1 ≈ 0.99, validation accuracy ≈ 0.75 (F1 ≈ 0.75, ROC-AUC ≈ 0.88), and final test accuracy ≈ 0.73 with weighted F1 ≈ 0.72 and ROC-AUC ≈ 0.87. This configuration[i] uses moderate depth, learning rate, subsampling, and L1/L2 regularization, which together reduce overfitting relative to the untuned version.[1]

7.2 Tranformer Models (Advanced model)

The transformer models reach respectable but slightly lower performance on the same problem. DistilBERT and BERT fine-tuning runs produce mid-to-high 60s to low 70s weighted F1 on validation, with ROC-AUC scores competitive but not clearly superior to tuned XGBoost and exhibit more variance across epochs and random seeds. [ii] While these models capture nuanced language patterns, their gains are constrained by limited dataset size, sequence

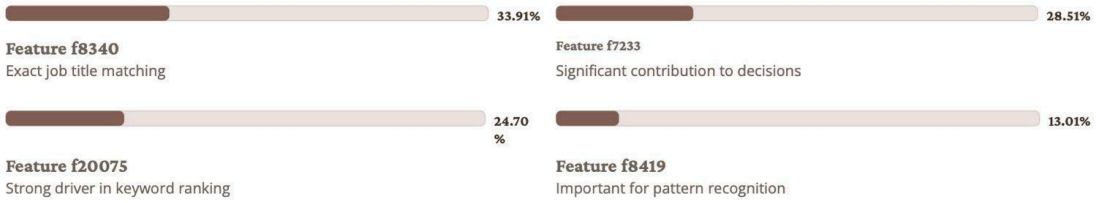
truncation, and the fact that strong hand-crafted similarity features already encode much of the matching signal.[3][1][2]

7.3 Comparisons

Model type	Model name	Accuracy (test)	Weighted F1 (test)	ROC-AUC (test)
Classical	Logistic Regression	≈ 0.49	≈ 0.46	lower 0.7s
Classical	Linear SVC	≈ 0.47	≈ 0.46	lower 0.7s
Classical	XGBoost (tuned)	≈ 0.73	≈ 0.72	≈ 0.87
Deep learning	DistilBERT (fine-tuned)	high-0.6s–0.7s	high-0.6s–0.7s	≈ 0.88
Deep learning	BERT-base (fine-tuned)	similar range	similar range	≈ 0.88

(Values for transformers are summarized ranges)[3][1]

The superior performance of the tuned XGBoost model is not just defined by its statistical metrics but also by the transparency and interpretability of its feature set. Unlike the opaque, raw-text input of the fine-tuned transformers, the classical pipeline allows for **detailed feature importance analysis**. This analysis confirmed that the model's decision-making was highly logical and driven by the manually engineered features. Diagnostics showed that signals like **exact job title matching** (e.g., Feature f8340) and various **keyword rankings** (e.g., Feature f20075) derived from the TF-IDF vectors and skill-matching mechanisms were the most impactful factors. This level of transparency is a critical advantage in an HR system, as it allows recruiters to audit why a candidate was flagged as a Good Fit or No Fit, which supports trust and compliance.



When we created the confusion matrices, they highlighted complementary error profiles. For the **tuned XGBoost model**, No Fit is predicted with high precision and recall, but there is **confusion between Potential Fit and No Fit**, reflecting the difficulty of borderline candidates. The best transformer model tends to **better distinguish Good Fit** from the other classes

(slightly higher recall for Good Fit), but still mislabels many Potential Fit resumes as No Fit and exhibits more off-diagonal mass overall.[3][1]

In terms of computational cost, XGBoost on 24k-dimensional sparse features trains in minutes on CPU, enabling rapid iteration and tuning. In contrast, DistilBERT/BERT fine-tuning requires GPU resources, longer training time per epoch, and careful batch-size and learning-rate choices; this cost is not insignificant in production or iterative research settings, especially given only modest performance gains in this dataset.[1][3]

8. Challenges and limitations

Despite rigorous cleaning, the underlying text still contains **synthetic artifacts** from job boards and templated recruiter language, which may cause models to latch onto patterns that do not generalize to organic resume–JD pairs. The **inability to reliably parse** fine-grained resume sections (Experience, Skills, Education, Certifications) led to dropping those fields; this likely removed structure that could have strengthened both feature-based and neural models.[2]

The 512-token context limit of BERT-style models **forces truncation** of long JDs and especially long resumes, meaning important skills or responsibilities near the end of documents are sometimes ignored. Finally, the “Potential Fit” label is inherently subjective and overlaps conceptually with both Good Fit and No Fit, making it the hardest class to separate; all models show heavy confusion around this boundary in their confusion matrices.[3][1][2]

9. Conclusions

Overall, the best approach for our project is a tuned XGBoost model built on TF-IDF features for JD and resume, plus engineered similarity and skill-overlap features, which slightly but consistently outperforms the more expensive transformer models on weighted F1 and ROC-AUC. This suggests that for medium-sized, relatively structured text-matching problems, strong classical baselines with thoughtful feature engineering remain highly competitive.[1][3]

9.1 Business Perspective

From a business perspective, a model tuned toward higher recall on Good Fit (as seen in the transformer runs) minimizes the risk of missing strong candidates, while a model tuned toward high precision on No Fit (as with XGBoost) can safely discard obvious mismatches and save recruiter time. For real-world deployment, these trade-offs should be balanced via decision

thresholds to align with the particular company or institution's tolerance for false positives versus false negatives.[3][1][2]

9.2 Future Scope

Future extensions include experimenting with long-context transformers like Longformer or BigBird to avoid truncation of long resumes, and augmenting the classical pipeline with NER-derived entities (universities, employers, tools) and structured features (years of experience, seniority).

10. References

1. modelling_classical.ipynb
2. Data-Cleaning-and-Preprocessing-Final.ipynb
3. Modelling_Advance_Final.ipynb

11. Appendix

11.1 Results of tuned XGBoost model:

11.1.1 PARAMETERS

subsample = 0.8
n_estimators = 400
max_depth = 6
learning_rate = 0.06
colsample_bytree = 0.7
reg_lambda = 2.5
reg_alpha = 0.4
min_child_weight = 3

11.1.2 TRAIN RESULTS

Accuracy: 0.9900
Weighted F1: 0.9900

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.99	0.99	1400
1	0.99	0.99	0.99	2800
2	1.00	0.98	0.99	1400

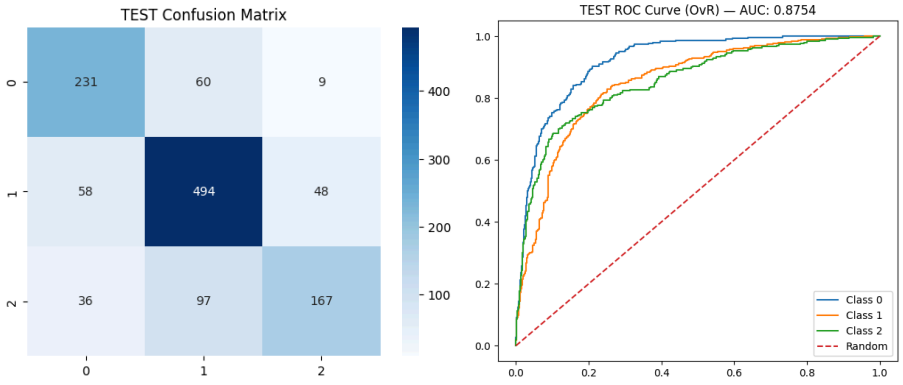
accuracy			0.99	5600
macro avg	0.99	0.99	0.99	5600
weighted avg	0.99	0.99	0.99	5600

11.1.3 TEST RESULTS

Accuracy: 0.7433
Weighted F1: 0.7390

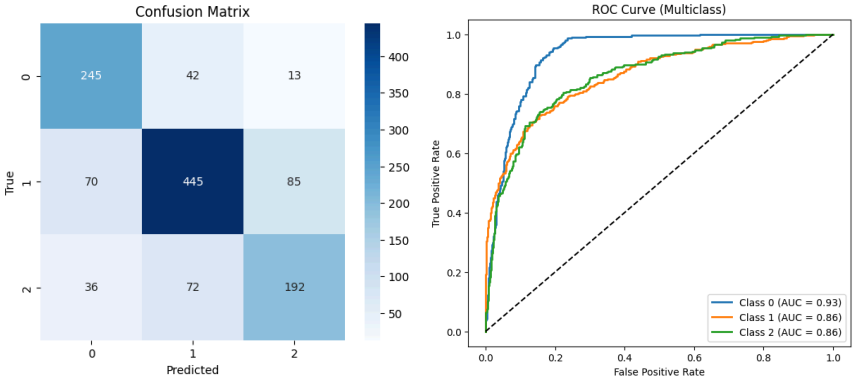
Classification Report:

	precision	recall	f1-score	support
0	0.71	0.77	0.74	300
1	0.76	0.82	0.79	600
2	0.75	0.56	0.64	300
accuracy			0.74	1200
macro avg	0.74	0.72	0.72	1200
weighted avg	0.74	0.74	0.74	1200



11.2 DistilBERT And BERT results

11.2.1 TEST RESULTS after rebuilding the dataset



11.2.2 TEST RESULTS with Cross Attention (BERT base Cross Attention)