UNIVERSITY OF PISA

Department of Information Engineering

Master's degree in Artificial Intelligence and Data Engineering

Data Mining and Machine Learning Project

Resume Classification Using Machine Learning

Work Group:
Andrea Bochicchio
Filippo Gambelli

ACADEMIC YEAR 2024/2025

Contents

1 Introduction

In today's job market, whenever a job offer is published, recruiters often receive hundreds of resumes. Manually reviewing each resume is a time-consuming task and can be affected by subjective judgment. This often results in potentially qualified candidates being overlooked simply because their profiles do not immediately stand out.

The primary objective of this project is to develop a system that automatically classifies resumes into predefined professional job categories based only on their textual content. This automated classification supports recruiting firms and Human Resources departments in efficiently organizing large volumes of resumes, accelerating the initial screening phase.

In addition to classification, we have also implemented a resume-job description matching system. Given a specific job description, the system identifies the most suitable candidates by measuring the textual similarity between the job offer and the resumes.

Automated systems for resume classification and matching have the potential to significantly improve recruitment workflows by reducing processing time, minimizing bias, and increasing the quality of hiring decisions. By automating the early stages of the selection process, recruiters can focus more on the most valuable aspects of their role: human interaction and the in-depth evaluation of top candidates.

In this project, we trained and evaluated several machine learning classifiers, using different feature extraction techniques to determine the most effective approach for resume classification. Our comparative analysis highlights the strengths and limitations of each method, providing insight into the design of robust AI-driven tools for real-world hiring scenarios.

The full implementation of this work is publicly available on GitHub: click here.

2 Datasets

To develop and evaluate our systems, we relied on two distinct datasets. The first one is a collection of resumes labeled with professional job categories, which was used for training and evaluating classification models. The second dataset consists of job descriptions collected from LinkedIn and it was used to implement and test the resume-job description matching functionality.

2.1 Resume Dataset

The first dataset (available here) is a collection of resumes obtained from the website LiveCareer.com. The dataset contains a total of 2,484 resumes, each labeled with one of 24 professional job categories. These categories are used as target classes for training and evaluating classification models.
Each resume is available in both PDF and plain text formats. PDF files are organized into subfolders, each named after the corresponding job category, and each file is identified by a unique filename. Plain text content was extracted through a preprocessing pipeline and stored in a CSV file, along with relevant metadata. Specifically, the CSV includes:

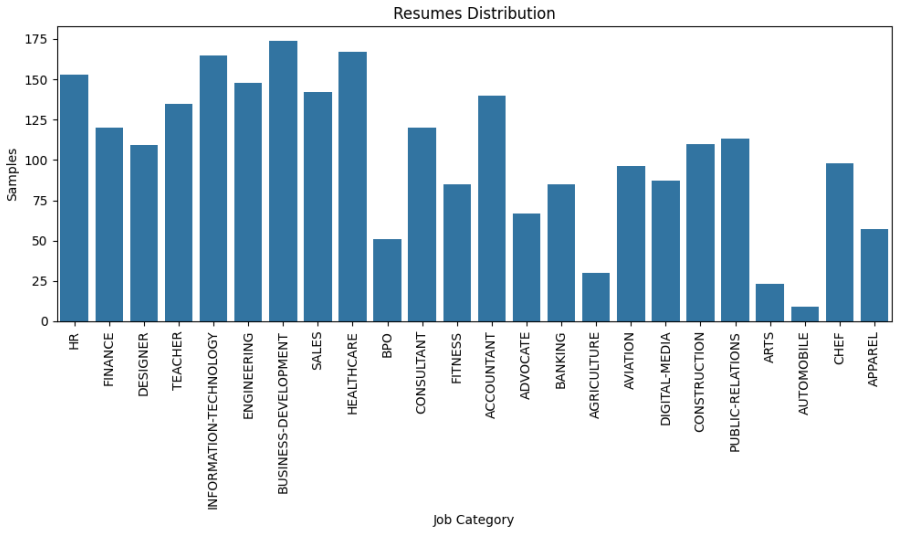
- `ID`: a unique identifier used as the filename of the corresponding PDF.
- `Resume_str`: the extracted plain text content of the resume.
- `Resume_html`: the original HTML structure of the resume, captured during web scraping.
- `Category`: the job category for which the resume was used to apply.

The 24 job categories included in the dataset are: *Business Development, Healthcare, Information Technology, HR, Engineering, Sales, Accountant, Teacher, Consultant, Finance, Public Relations, Construction, Designer, Chef, Aviation, Digital Media, Fitness, Banking, Advocate, Apparel, BPO, Agriculture, Arts, Automobile.*

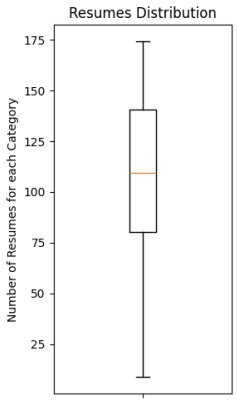
2.1.1 Visualization

To gain a deeper understanding of the resume dataset, we performed an exploratory analysis of the class distribution. Figure 2.1a shows a histogram representing the number of resumes available for each job category, while Figure 2.1b displays a box plot summarizing the same distribution.
From these visualizations, it is evident that the dataset is significantly unbalanced. Some categories have only a few samples, such as Agriculture, Arts, and Automobile, whereas others like Business Development and Healthcare contain more examples. To address this issue, before training any models, we applied appropriate data balancing techniques.

In addition to analysing class distribution, we also examined the distribution of word counts across the resumes in the dataset. Understanding the length of resumes can provide valuable insight into the variability of document size, which may influence feature extraction and classification performance.
Figure 2.2 illustrates the number of words per resume using both a histogram and a box plot. As shown in the histogram, most resumes contain between 250 and 1,500 words. As highlighted in the box plot (Figure 2.2b), one resume has a word count of zero, indicating that the corresponding text field is empty. This represents a data quality issue that we address during the preprocessing phase (see Section 3.1).

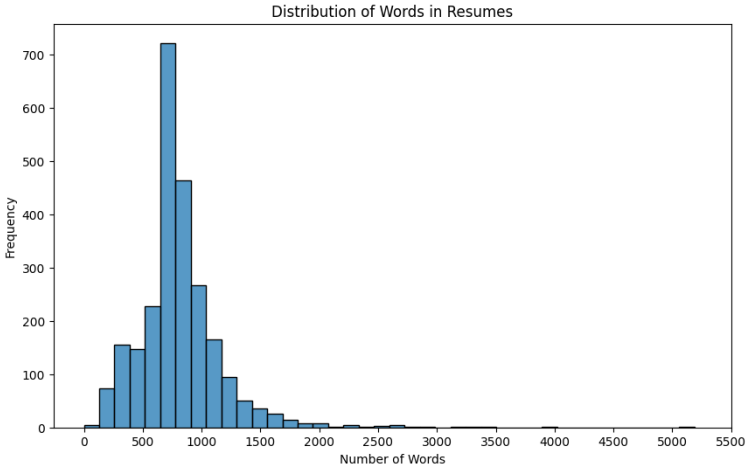


(a) Class Distribution Histogram

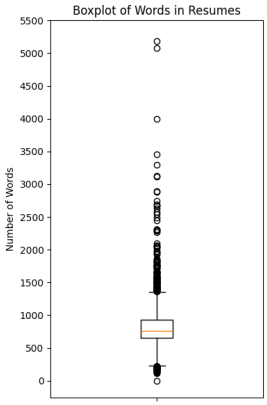


(b) Class Distribution Box Plot

Figure 2.1: Class distribution



(a) Word Count Histogram



(b) Word Count Box Plot

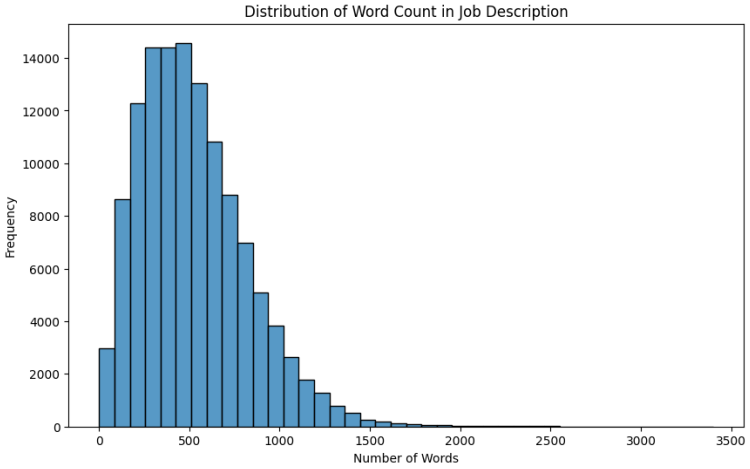
Figure 2.2: Distribution of word counts

2.2 Job Description Dataset

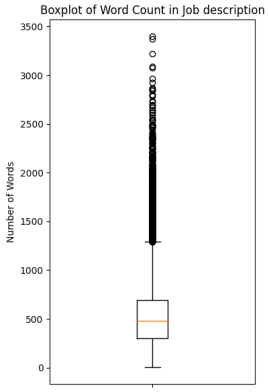
To implement and evaluate the matching functionality of our system, where job postings are compared against resumes to identify the best candidates, we used a separate dataset (available here) of job postings collected from LinkedIn during 2023 and 2024. This dataset consists of 123,849 entries, each containing a job title, a detailed description of the role, and many other details.
These job descriptions provide the textual data necessary for computing similarity scores with candidate resumes, enabling the system to recommend the most suitable matches for each job offer.

2.2.1 Visualization

To better understand the characteristics of this dataset, we generated two visualizations: a histogram and a box plot (Figure 2.3) illustrating the distribution of the number of words in the job descriptions. These visualizations help reveal the typical length of job postings and guide preprocessing and modelling decisions.



(a) Word Count Histogram



(b) Word Count Box Plot

Figure 2.3: Distribution of word counts

3 Data Preprocessing

Data preprocessing is a critical step in any data mining and machine learning pipeline. It ensures that raw data are transformed into a clean, consistent, and informative format that algorithms can effectively learn from. Proper preprocessing helps reduce noise, handle missing or invalid entries, normalize variations in text, and ultimately improves model accuracy and convergence speed. In this chapter, we present the preprocessing methods applied to our two datasets.

Our preprocessing workflow consists of the following main phases:

1. **Data cleaning**: removal of irrelevant or invalid records and fields.
2. **Text normalization**: transforming text into a consistent form.
3. **Tokenization and filtering**: splitting text into tokens and removing undesired units.
4. **Morphological processing**: applying lemmatization and/or stemming to reduce words to their base forms.
5. **Dataset splitting**: dividing data into training and test sets with stratified sampling.
6. **Parameter exploration**: generating multiple preprocessed variants to evaluate the impact of different settings.

3.1 Resume Preprocessing

As already mentioned before (see Section 2.1) the resumes dataset contains the following columns: `ID`, `Resume_str`, `Resume_html`, `Category`. For our classification tasks, the HTML representation is not needed. Therefore, as a first step, we removed the `Resume_html` column from the dataset.
During initial inspection (see Figure 2.1b), we discovered one record whose `Resume_str` field was an empty string. Since it provided no useful features for training, we dropped that row from the dataset.

3.1.1 Text Preprocessing Function

We implemented a reusable `preprocess_text` function in the `PreProcessing.ipynb` notebook. Its signature is:

```
1  def preprocess_text(text: str, remove_stopwords: bool = True,
2      use_lemmatization: bool = True, use_stemming: bool = False) -> List[str]
```

This function performs the following operations:

- Convert text to lowercase.
- Remove punctuation by retaining only letters and numeric digits.
- Remove underscore characters and collapse multiple whitespace into a single space.
- Tokenize the cleaned text into individual words.
- Discard tokens containing numerical digits (e.g., 'C++17' or '2021').
- Remove stopwords based on the `remove_stopwords` flag (e.g., 'the', 'and', 'is').
- Apply lemmatization or stemming to each token according to `use_lemmatization` and `use_stemming` flags.

Lemmatization vs. Stemming Both lemmatization and stemming aim to reduce inflected or derived words to a common base form, but they differ in approach:

- **Stemming** uses crude heuristic rules to strip affixes. For example, the Porter stemmer reduces "running", "runs", and "ran" all to "run". Stemming can produce non-dictionary forms (e.g., "studies" → "studi").
- **Lemmatization** uses vocabulary and morphological analysis to map words to their dictionary (lemma) forms. For instance, "better" → "good", "geese" → "goose", and "running" → "run". Lemmatization preserves valid words but requires part-of-speech information to choose the correct lemma.

In our experiments, we compared both approaches to identify which combinations of feature extraction techniques and classifiers yield the best performance.

3.1.2 Train/Test Split

We split the dataset into a training set (80%) and a test set (20%) prior to model training. The `text preprocessing` function, described in the previous section, was then applied to both subsets. This ensured that preprocessing was consistently performed across all data used for training and final evaluation.

3.2 Job Description Preprocessing

The preprocessing of the job descriptions dataset followed a similar approach to the one used for the resumes. However, a few additional steps were necessary due to the size and structure of the data.
As a first step, we removed all fields that were not directly relevant to our matching objectives. From the original table, only the following three columns were retained:

- `ID`: unique identifier for each job posting,
- `Title`: the job announcement title,
- `Description`: the full text of the job description.

We then inspected the filtered dataset for missing or malformed entries. Of the 123,842 records, seven contained a `NULL` value in the `Description` field. Since these entries could not contribute any textual features, they were dropped from the dataset.
To ensure that only sufficiently detailed descriptions remained, we computed the word count of each `Description`. We discarded all records whose description contained 20 words or fewer. This threshold step reduced noise from overly brief postings and retained only those descriptions likely to contain meaningful, discriminative content.
Finally, we applied the same `preprocess_text` function introduced in Section 3.1.1 to the cleaned and filtered job descriptions.
Unlike the resume dataset, we did not split the job descriptions into training and test sets, since the development and evaluation of the matching system did not require a separate test dataset.

4 Text Representation

In natural language processing (NLP), the way textual data is transformed into numerical features greatly affects the performance of machine learning models. For the resume classification task, we experimented with four different text representation techniques: **TF-IDF**, **Word2Vec** (W2V), **Doc2Vec** (D2V) and **Sentence-BERT** (SBERT). Each approach captures different aspects of textual semantics and structures.

4.1 TF-IDF

Term Frequency-Inverse Document Frequency is a statistical method used to convert raw text into numerical feature vectors. It helps quantify how important a word is to a specific document, relative to a collection (corpus) of documents. The key idea behind TF-IDF is to assign higher weight to words that are frequent in a document but rare across the entire corpus, as these words tend to carry more meaningful and discriminative information.
TF-IDF consists of two main components:

1. **Term Frequency** (TF) measures how frequently a word appears in a document. It is computed as:

$$\text{TF}(t, d) = \frac{f_{t,d}}{L_d}$$

   where:

   - $f_{t,d}$ = number of times term t appears in document d
   - L_d = total number of terms in document d

   This gives a normalized frequency for each term in a document.

2. **Inverse Document Frequency** (IDF) measures how rare or unique a word is across all documents in the corpus. It is calculated as:

$$\text{IDF}(t) = \log\left(\frac{N}{1 + n_t}\right)$$

   where:

   - N = total number of documents
   - n_t = number of documents containing term t

   The "+1" in the denominator is used to prevent division by zero. Common words that appear in many documents (e.g., "the", "is", "and") have low IDF values, while rare words have higher values.

The **final TF-IDF score** is the product of TF and IDF:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

This means that a term will receive a high TF-IDF score if:

- It occurs frequently in a specific document (high TF),
- But rarely occurs across the entire corpus (high IDF).

This representation allows the model to focus on terms that are both representative of the document and discriminative among classes.

In this project, we used `TfidfVectorizer` from Scikit-learn to transform resumes into TF-IDF vectors. We also applied filters to remove extremely rare or overly common terms, and captured multiple n-gram patterns (from unigrams to trigrams):

```
tfidf_vect = TfidfVectorizer(
    lowercase=True,          # Convert all text to lowercase
    binary=False,            # Use actual term frequency values
    max_features=10000,      # Limit the vocabulary size to 10,000
    ngram_range=(1, 3),      # Extract unigrams, bigrams, and trigrams
    max_df=0.8,              # Ignore terms that appear in more than 80% of documents
    min_df=2                 # Ignore terms that appear in fewer than 2 documents
)
```

4.2 Word2Vec

Word2Vec is a neural embedding technique that maps words to dense vectors such that semantically similar words are close in the vector space. Unlike TF-IDF, which represents words as sparse vectors based on frequency, embeddings capture semantic relationships between words based on their context.

```
advancedW2V = AdvancedW2VTransformer(
    vector_size=50,          # Dimensionality of the document vectors
    window=10,               # Context window size
    min_count=2,             # Minimum word frequency threshold
    epochs=40                # Number of training epochs
)
```

To represent an entire document, we compute a weighted average of the vectors of the words it contains. This aggregation step is essential because embeddings are defined at the word level, and without proper aggregation (e.g., using simple averaging or TF-IDF-weighted averaging), we would lose the overall semantic representation of the document.
In this project, we enhanced Word2Vec by incorporating TF-IDF weights into the aggregation of word vectors, producing more informative document embeddings.

```
def _get_doc_vector(self, tokens, tfidf_row):
    word2idx = self.tfidf.vocabulary_
    vecs, weights = [], []
    for word in tokens:
        if word in self.w2v_model.wv and word in word2idx:
            vecs.append(self.w2v_model.wv[word])
            weights.append(tfidf_row[0, word2idx[word]])
    if not vecs:
        return np.zeros(self.vector_size)
    vecs = np.array(vecs)
    w = np.array(weights)
    w = w / w.sum() if w.sum() > 0 else np.ones_like(w) / len(w)
    stats = [
        np.average(vecs, axis=0, weights=w)
    ]
    return np.concatenate(stats)
```

4.3 Doc2Vec

Doc2Vec extends Word2Vec by generating vector representations for entire documents rather than individual words. It captures the semantic context of documents directly, without needing to aggregate word vectors.

```
d2v_model = AdvancedD2VTransformer(
    vector_size=200,  # Dimensionality of the document vectors
    window=10,        # Context window size
    min_count=2,      # Minimum word frequency threshold
    dm=1,             # Use Distributed Memory (DM); use 0 for DBOW
    epochs=20         # Number of training epochs
)
```

4.4 Sentence-BERT

SBERT is a transformer-based model that provides semantically meaningful sentence or document embeddings. It modifies BERT using a Siamese network structure to efficiently produce embeddings that are suitable for semantic similarity tasks.
Unlike TF-IDF and Word2Vec, SBERT captures deep contextual and semantic information across entire sentences or paragraphs.
BERT (Bidirectional Encoder Representations from Transformers), the foundation of SBERT, is a language model developed by Google that understands the meaning of words based on their surrounding context in both directions. This makes it highly effective at understanding language on a deeper, more nuanced level than traditional models.
However, vanilla BERT is not optimized for producing sentence-level embeddings suitable for similarity comparisons. That's why SBERT extends BERT by introducing a Siamese network structure that enables efficient and meaningful comparison between textual inputs.
In the context of CV (curriculum vitae) analysis, SBERT is particularly effective because CVs are typically composed of full sentences or short descriptive paragraphs. These require a model that can understand the overall semantic meaning of phrases.

```
sbert_model = SentenceTransformer('all-MiniLM-L12-v2')
```

5 Resume Classification

In this chapter, we detail the complete pipeline employed to build, train, and evaluate models designed for the resume classification task. The chapter is organized into three main sections. First, we provide a detailed theoretical overview of the machine learning algorithms chosen for the classification task: Logistic Regression, Random Forest, and Support Vector Machine (SVM).
Next, we explain how these models were integrated with the four feature extraction techniques discussed in Chapter 4, resulting in multiple combinations of input representations and classifiers.
Finally, we describe the model training and evaluation workflow, which includes the use of k-fold cross-validation to ensure robust performance estimates, grid search for hyperparameter tuning, and the application of evaluation metrics such as accuracy, precision, recall, and F1-score. In addition to metric-based comparisons, we also performed statistical significance testing to assess whether performance differences between models were meaningful. This comprehensive approach allowed us to identify the best-performing model configuration for the task of resume classification.

5.1 Models

In this section, we present the classification models used to predict the category of a resume based on its textual content. The objective of this task is to build supervised models capable of learning patterns from labeled resume data and generalizing to unseen examples.
We selected and compared three widely used machine learning algorithms:

- **Logistic Regression** (LR)
- **Random Forest** (RF)
- **Support Vector Machine** (SVM)

5.1.1 Logistic Regression

Logistic Regression is a model commonly used for binary and multiclass classification. Unlike linear regression, which predicts a continuous value, logistic regression estimates the probability that a given input belongs to a particular class.
For binary classification, the model is defined as:

$$P(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}$$

where:

- $\mathbf{x} \in \mathbb{R}^n$ is the input feature vector,
- $\mathbf{w} \in \mathbb{R}^n$ is the weight vector,
- $b \in \mathbb{R}$ is the bias term,
- $\sigma(\cdot)$ is the sigmoid function that maps real values to probabilities between 0 and 1.

For multiclass classification, the softmax function generalizes the sigmoid function. Given K classes, the model estimates the probability of class k as:

$$P(y = k \mid \mathbf{x}) = \frac{e^{\mathbf{w}_k^\top \mathbf{x} + b_k}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^\top \mathbf{x} + b_j}}$$

Logistic Regression is trained by minimizing the cross-entropy loss, which measures the distance between the predicted probabilities and the true labels.

5.1.2 Random Forest

Random Forest is an ensemble learning method that builds a collection of decision trees and aggregates their outputs. It combines the simplicity of decision trees with improved generalization by reducing variance.
A Random Forest classifier works by:

- Training each decision tree on a random subset (with replacement) of the training data (bootstrap sampling),
- Selecting a random subset of features at each split in each tree to reduce correlation between trees,
- Aggregating predictions by majority vote (for classification) across all trees in the forest.

Given T trees $\{h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_T(\mathbf{x})\}$, the final prediction is:

$$\hat{y} = \text{mode}\{h_1(\mathbf{x}), h_2(\mathbf{x}), \ldots, h_T(\mathbf{x})\}$$

Random Forest is known for its robustness to overfitting, especially on large datasets with many noisy features, and it requires little feature scaling or preprocessing.

5.1.3 Support Vector Machine

Support Vector Machines (SVMs) are powerful classifiers that aim to find the optimal hyperplane that maximally separates data points of different classes. For linearly separable data, the decision boundary is defined by:

$$\mathbf{w}^\top \mathbf{x} + b = 0$$

SVM finds the hyperplane that maximizes the margin, which is the distance between the hyperplane and the nearest data points (called support vectors). The optimization problem for a hard-margin SVM is:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to } y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 \quad \forall i$$

For non-linearly separable data, a soft-margin SVM introduces slack variables ξ_i and a penalty parameter C to allow some misclassifications:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i \quad \text{subject to } y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i$$

SVMs can also handle non-linear relationships using the kernel trick, which maps the input data into a higher-dimensional space. Common kernels include:

- Linear kernel: $K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^\top \mathbf{z}$
- Polynomial kernel: $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z} + c)^d$
- Radial Basis Function (RBF): $K(\mathbf{x}, \mathbf{z}) = \exp(-\gamma\|\mathbf{x} - \mathbf{z}\|^2)$

SVMs are especially effective in high-dimensional spaces and are widely used for text classification tasks.

5.2 Model Training

To thoroughly evaluate how different feature representations affect model performance, we tested all three classifiers — Logistic Regression, Random Forest, and Support Vector Machine — in combination with each of the four feature extraction methods described in Chapter 4. This resulted in a total of twelve model–feature extraction configurations [1].
The process was structured as follows:

1. For each of the twelve model–feature extraction combinations, we ran a grid search with cross-validation to identify the optimal hyperparameters. This ensured that each configuration was trained under the best possible settings.

2. For each classifier, we compared the performance of its four optimized configurations, one for each feature extraction method, to select the best-performing variant.

3. We conducted a comprehensive comparison among the three best models (one per classifier) in order to identify the overall top-performing approach for resume classification.

Evaluation was based on multiple performance metrics, including accuracy, precision, recall, and F1-score. In addition, we applied statistical significance tests to verify whether the observed differences between models were meaningful and not due to random variation in the data splits.
This systematic and rigorous approach allowed us to gain deeper insights into the interaction between feature representation and model choice, and ensured a fair and data-driven selection of the final classification model.

5.2.1 Pipeline Design

In this project, we design a robust and reproducible pipeline for the task of resume classification. Each configuration, combining feature extraction and classification, follows a consistent and modular workflow. Specifically, we construct a machine learning pipeline consisting of a feature extraction step, followed by class balancing using SMOTE, and finally, model training. This entire pipeline is then passed to a `GridSearchCV` procedure to perform an exhaustive search over the hyperparameter space for each combination of feature representation and classifier.

Dataset Split Prior to model training (see Section 3.1.2), we prepared our data by loading a CSV file that already contains 80% of the original dataset. We reserved the remaining 20% as a hold-out test set, which remains untouched until final evaluation. This approach ensures an unbiased assessment of model generalization and makes efficient use of preprocessed data artefacts.

Differentiated Text Preprocessing Because feature-extraction techniques respond differently to text normalization, we use two different preprocessing approaches:

- For **TF–IDF**, **Word2Vec** and **Doc2Vec**:

    1. Convert to lowercase, ensuring case-insensitive term matching.
    2. Strip punctuation and underscores to avoid spurious tokens.
    3. Collapse repeated whitespace into single spaces for consistent token boundaries.
    4. Remove stop-words to eliminate common but uninformative terms, reducing noise and vector dimensionality.

[1] The full implementation for these experiments is available in the GitHub repository. There are three main folders named after the classifiers (`LogisticRegression`, `RandomForest`, and `SupportVectorMachine`). Each folder contains four Jupyter notebooks, one for each feature extraction method.

5. Apply lemmatization so that variants of a word share a single canonical form, improving embedding stability.

- For **Sentence-BERT** (SBERT):

  1. Convert to lowercase to align with model vocabulary casing.
  2. Remove punctuation and underscores to prevent tokenization inconsistencies.
  3. Collapse repeated whitespace.

  We intentionally avoid stop-word removal and lemmatization for SBERT because:

  – SBERT's pretraining on massive, varied corpora has already learned to weigh common function words appropriately in context.

  – Aggressive preprocessing can strip subtle linguistic cues (e.g., negations, prepositions) that SBERT's transformer architecture uses to build deep semantic representations.

  – Minimal cleaning preserves the natural sentence structure and word usage patterns, allowing the model to leverage its full contextual understanding without introducing preprocessing artifacts.

Feature Extraction Resumes are unstructured text and require transformation into numerical representations. As already mention in the Chapter 4, we compare:

- TF–IDF (Term Frequency–Inverse Document Frequency)
- Word2Vec (W2V)
- Doc2Vec (D2V)
- Sentence-BERT (SBERT)

Handling Class Imbalance As noted in Section 2.1.1, our resume dataset is imbalanced: some categories contain a large number of samples, while others have only a few. To address this, we employ oversampling rather than undersampling, preserving all available data from majority classes.
We use the *Synthetic Minority Over-sampling Technique* (**SMOTE**), which generates new synthetic examples for minority classes by interpolating between existing samples in feature space. Concretely, for each minority example x_i, SMOTE selects one of its k nearest neighbors x_{nn} (under Euclidean distance) and creates a new point.
This approach produces more diverse synthetic samples than simple replication, reducing overfitting while improving classifier robustness.
To avoid over-generating synthetic data, we apply SMOTE only to those classes whose current support is less than 75% of the mean class size. We compute:

$$\overline{|D|} = \frac{1}{C}\sum_{c=1}^{C}|D_c|, \quad \mathcal{M} = \{\, c : |D_c| < 0.75\,\overline{|D|}\,\},$$

and set the target size for each minority class $c \in \mathcal{M}$ as

$$\text{target\_size} = 0.75 \times \max_{c'} |D_{c'}|.$$

This custom sampling strategy balances the dataset without introducing excessive synthetic examples.

Here is the code:

```
class_counts = df['Category'].value_counts()
min_classes = class_counts[class_counts < 0.75*class_counts.mean()].index.tolist()

print(f"Applying SMOTE to classes: {min_classes}")

target_size = int(0.75*class_counts.max())
sampling_strategy = {cls: target_size for cls in min_classes}
```

Cross-Validation and Hyperparameter Tuning To obtain reliable performance estimates without data leakage, we use stratified 5-fold cross-validation, which preserves class proportions in each fold:

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Within each fold, SMOTE is applied only to the training partition, ensuring that no synthetic samples leak into validation sets. For example, the TF–IDF + Random Forest pipeline is defined as:

```
pipeline = Pipeline([
    ('tfidf', tfidf_vect),
    ('smote', SMOTE(sampling_strategy=sampling_strategy, random_state=42, k_neighbors=4)),
    ('clf', RandomForestClassifier(random_state=42, n_jobs=-1))
])
```

Hyperparameter configurations for all model–feature-extraction combinations were explored extensively in our GitHub repository. For instance, for the Random Forest classifier, the parameter grid is:

```
param_grid = {
    'clf__n_estimators': [400, 700],
    'clf__max_features': ['sqrt', 'log2'],
    'clf__max_depth' : [8, 9, 10],
    'clf__criterion' : ['gini', 'entropy'],
    'clf__min_samples_split': [10],
    'clf__min_samples_leaf': [15]
}

scoring = ['accuracy', 'f1_weighted', 'precision_weighted', 'recall_weighted', 'roc_auc_ovr_weighted']

grid = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=skf,
    scoring= scoring,
    refit = False,
    return_train_score=True,
    n_jobs=-1
)
```

We evaluate models using a suite of metrics—accuracy, weighted precision, weighted recall, weighted F1-score, and weighted ROC AUC—to capture different aspects of performance under class imbalance.

Model Evaluation and Visualization For each hyperparameter combination, we record fold-wise training and validation scores. We then compute mean and standard deviation across folds, visualizing the results via:

- *Error-bar plots* showing mean ± std for train vs. validation, highlighting generalization gaps.
- *Boxplots* of fold-wise scores, revealing score dispersion and potential outliers.

These visual diagnostics guide the selection of a model that balances overall accuracy with stability across folds.

5.3 Model Selection Strategy

To select the best overall classification model, we followed a rigorous two-stage evaluation strategy combining performance metrics and statistical significance testing. This process was designed to ensure that our final model was not only accurate, but also robust and generalizable.

5.3.1 Stage 1: Per-Classifier Optimization

In the first stage, we focused on identifying the best model configuration for each of the three classifiers by evaluating their performance across all four feature extraction methods.
To select the best configuration within each combination, we used the weighted F1-score as the primary criterion. The F1-score, which is the harmonic mean of precision and recall, is particularly informative in imbalanced classification scenarios, as it captures the trade-off between false positives and false negatives. In addition to maximizing the F1-score, we ensured that the selected model did not exhibit signs of overfitting. This was evaluated by analysing the gap between training and validation scores: models that showed unusually high performance on the training set but poor generalization on validation data were discarded in favour of more stable alternatives.
This process resulted in twelve optimized models (one per classifier–feature pair). From these, we selected the best-performing model for each classifier, again based primarily on F1-score. This led us to identify the optimal feature extraction method for each classifier.

5.3.2 Stage 2: Cross-Classifier Comparison

Interestingly, the best-performing configuration for all three classifiers used the same feature extraction method: Sentence-BERT (SBERT). This result is aligned with expectations for the resume classification domain. SBERT, being a transformer-based language model pretrained on large corpora for semantic similarity tasks, is particularly effective in capturing nuanced meaning in short to medium length text like resumes. Unlike bag-of-words or shallow embedding methods (e.g., TF-IDF, Word2Vec, Doc2Vec), SBERT generates contextualized vector representations at the sentence level, preserving semantic relationships and discarding irrelevant syntactic variation—both of which are crucial for accurately modelling job-related information.
Figure 5.1 shows the performance comparison of the Support Vector Machine (SVM) classifier across the four different feature extraction methods. Similarly, Figure 5.2 illustrates the same comparison for Logistic Regression, while Figure 5.3 presents the results for Random Forest. These visual comparisons highlight the consistent superiority of SBERT over traditional feature representations across all classifiers, reinforcing its effectiveness for this classification task.

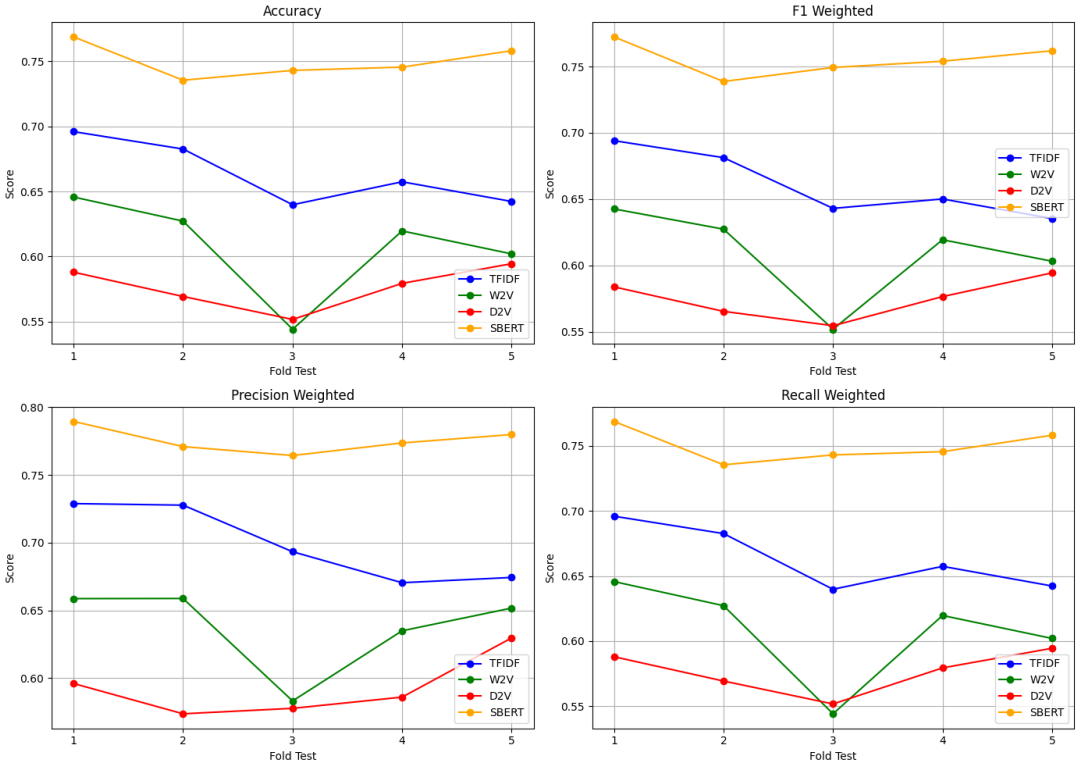


Figure 5.1: Per-fold performance of the SVM classifier across different feature representations

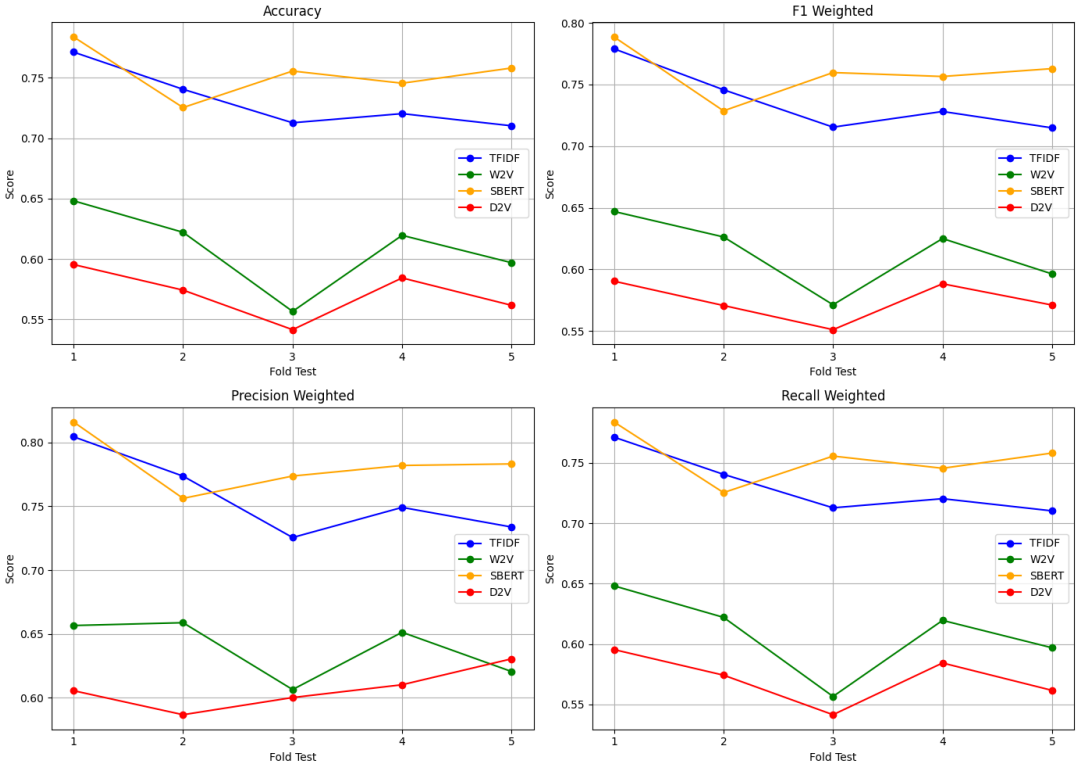


Figure 5.2: Per-fold performance of the Logistic Regression classifier across different feature representations

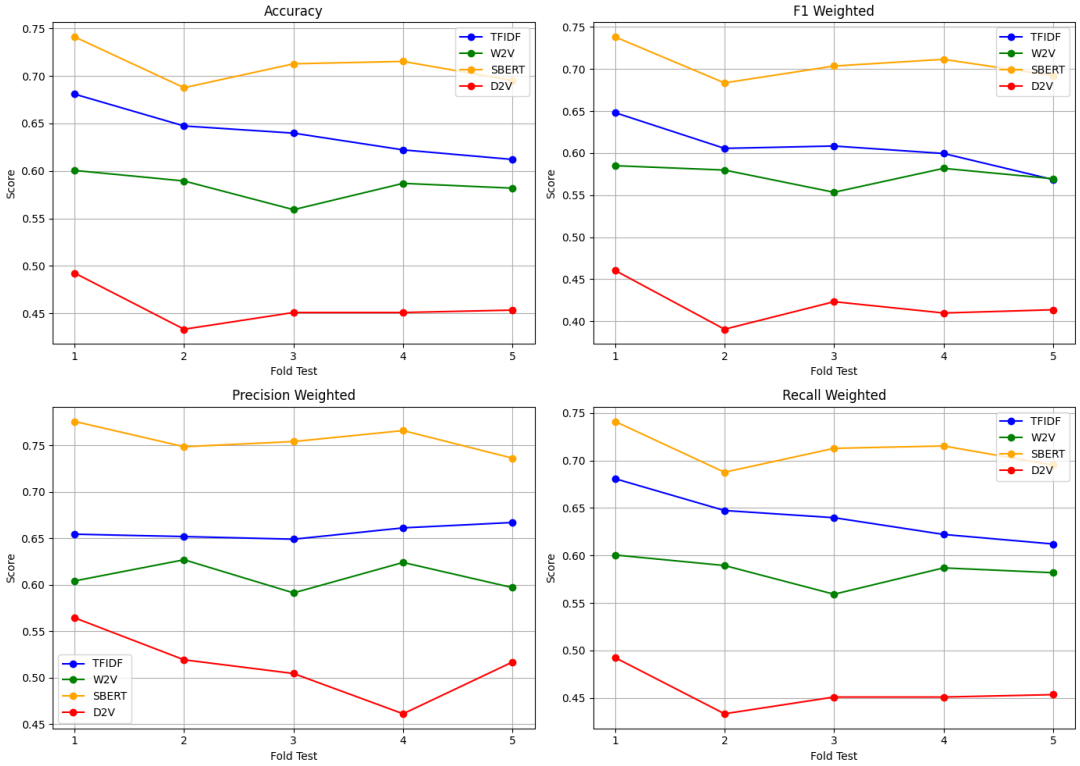


Figure 5.3: Per-fold performance of the Random Forest classifier across different feature representations

With SBERT established as the common representation, we then compared the three best classifier configurations all using SBERT embeddings.

5.3.3 Statistical Significance Testing

To assess whether the performance differences between these three top models were statistically meaningful, we conducted pairwise statistical significance testing using the Wilcoxon signed-rank test.
The Wilcoxon signed-rank test is a non-parametric alternative to the paired t-test, used when the assumption of normality cannot be guaranteed. It tests whether the median difference between paired observations is zero, making it suitable for comparing two models across multiple cross-validation folds. Specifically, for each pair of models, we computed fold-wise differences in F1-score and AUC (Area Under the ROC Curve), and used the Wilcoxon test to determine whether these differences were statistically significant.
The F1-score, as previously discussed, balances precision and recall, making it ideal for imbalanced datasets. The AUC, or Area Under the Receiver Operating Characteristic Curve, measures a model's ability to distinguish between classes. AUC values range from 0.5 (no discrimination) to 1.0 (perfect discrimination), and are especially useful when evaluating classifiers under varying classification thresholds. Our Wilcoxon tests yielded p-values greater than the standard significance level ($\alpha = 0.05$) for all model pairs and for both F1-score and AUC. This indicated that none of the models significantly outperformed the others from a statistical point of view.

5.3.4 Final Model Choice

Given the lack of statistically significant differences, we proceeded to choose the best model based on a comprehensive comparison of standard performance metrics: accuracy, F1-score, precision, and recall. These metrics were aggregated across the 5 cross-validation folds. We visualized the performance in two ways:

- **Per-fold metric plots:** Each model's performance metrics were plotted with error bars (mean ± standard deviation) to highlight consistency across folds (see Figure 5.4).
- **Average performance histogram:** We also plotted a bar chart summarizing the mean values of the four metrics for each model (see Figure 5.5).

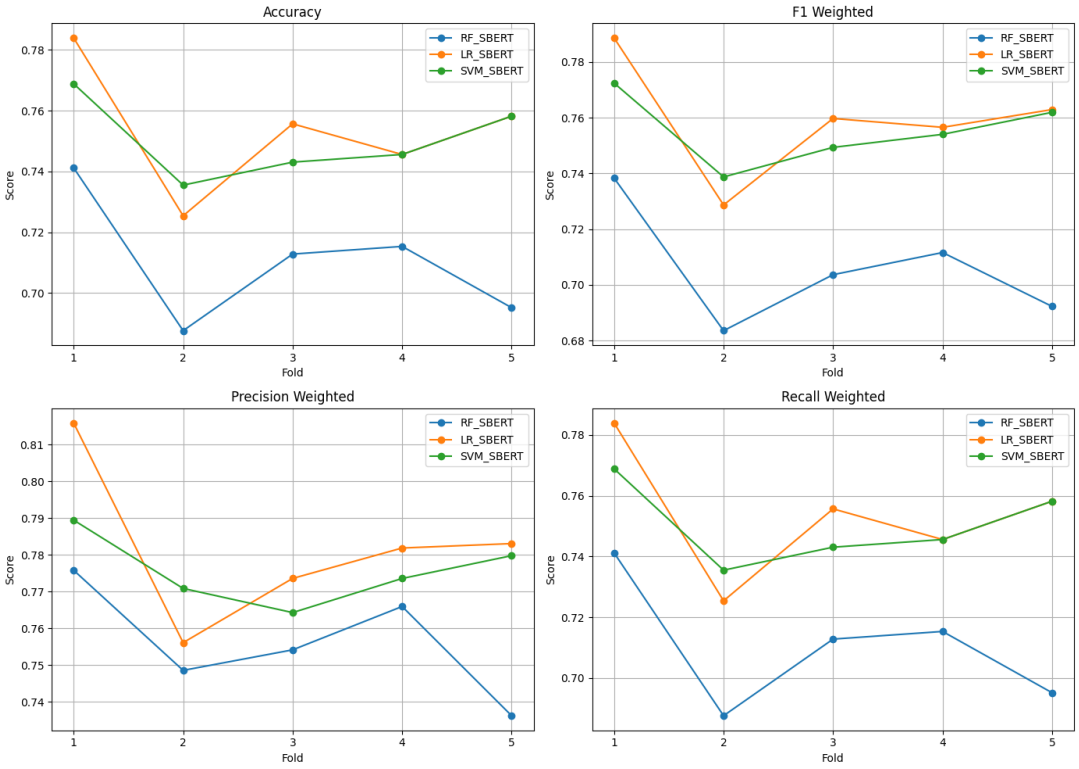


Figure 5.4: Per-fold metric plots

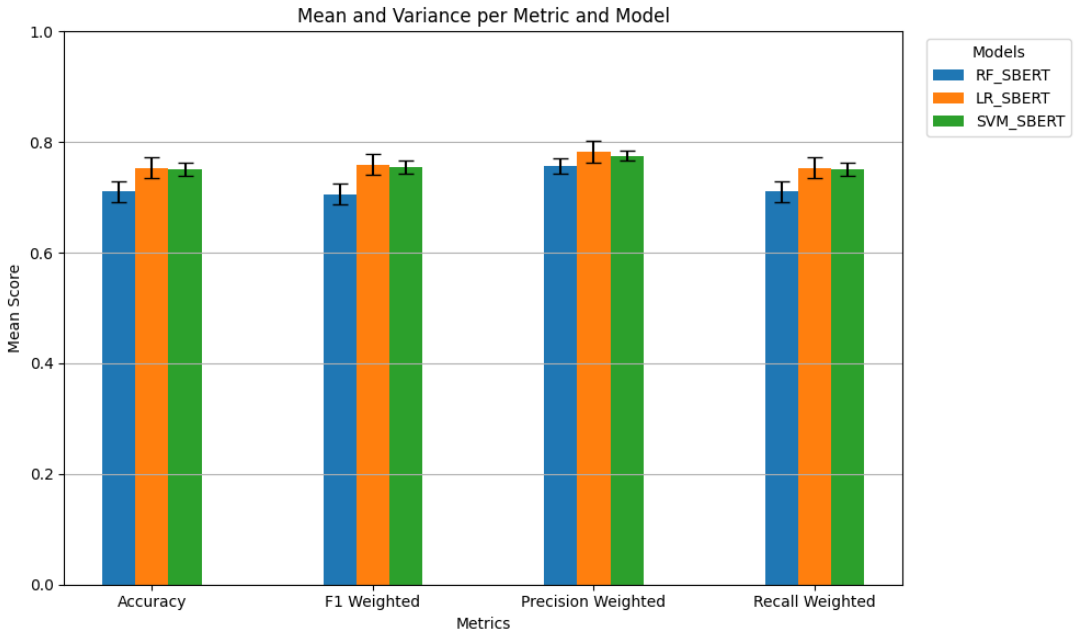


Figure 5.5: average performance histogram

From these comparisons, we found that the Random Forest model performed the worst across all metrics.

In contrast, the Logistic Regression and Support Vector Machine models had very similar average scores. To break the tie, we examined the variance of each metric across folds. The model with the smallest variance, indicating more consistent performance, was chosen as the final model.
Based on this analysis, the **Support Vector Machine** with **SBERT embeddings** was selected as the final model for resume classification, due to its strong overall performance and lower variance across evaluation folds.

5.4 Final Evaluation

After selecting the best classification model — Support Vector Machine (SVM) trained on SBERT embeddings — we conducted a final evaluation on completely unseen data. This step aimed to provide an unbiased estimate of the model's generalization performance.

5.4.1 Evaluation Setup

Until this point, model training and selection were carried out exclusively on 80% of the original dataset using 5-fold cross-validation. The remaining 20% of the data was kept aside as a held-out test set and was never used during training, hyperparameter tuning, or validation.
For the final evaluation, we trained the selected model on the entire 80% training set using the same preprocessing pipeline adopted throughout the project. In particular:

- SBERT embeddings were used to convert resumes into semantic vector representations.
- The dataset was imbalanced across classes, so we applied the SMOTE (Synthetic Minority Oversampling Technique) algorithm to the training set to synthetically augment the minority classes.
- Unlike previous phases, no cross-validation or hyperparameter tuning (e.g., grid search) was performed at this stage. We used a fixed training and test set split.

5.4.2 Performance Results

Once trained on the full 80% portion of the dataset, the selected SVM model was evaluated on the previously unseen 20% test set. The results clearly show that the model maintains strong generalization capabilities. In fact, the **test accuracy** remains high at **0.7586**, while the **training accuracy** is **0.8646**. This indicates a good balance between learning and overfitting.
Beyond overall accuracy, we also report precision, recall, and F1-score for each class, which provide a more nuanced view of performance, especially in the context of our multi-class and imbalanced classification problem. The following table summarizes the detailed classification report:

Class	Precision	Recall	F1-Score	Support
ACCOUNTANT	0.90	0.93	0.91	28
ADVOCATE	0.56	0.69	0.62	13
AGRICULTURE	1.00	0.50	0.67	6
APPAREL	0.44	0.64	0.52	11
ARTS	1.00	0.40	0.57	5
AUTOMOBILE	0.00	0.00	0.00	2
AVIATION	0.65	0.79	0.71	19
BANKING	0.78	0.41	0.54	17
BPO	0.30	0.60	0.40	10
BUSINESS-DEVELOPMENT	0.73	0.69	0.71	35
CHEF	0.90	0.95	0.93	20
CONSTRUCTION	0.95	0.86	0.90	22
CONSULTANT	0.73	0.79	0.76	24
DESIGNER	0.86	0.86	0.86	22
DIGITAL-MEDIA	0.65	0.65	0.65	17
ENGINEERING	0.83	0.80	0.81	30
FINANCE	0.81	0.88	0.84	24
FITNESS	1.00	0.76	0.87	17
HEALTHCARE	0.69	0.73	0.71	33
HR	0.88	0.71	0.79	31
INFORMATION-TECHNOLOGY	0.78	0.76	0.77	33
PUBLIC-RELATIONS	0.62	0.57	0.59	23
SALES	0.83	0.86	0.84	28
TEACHER	0.83	0.93	0.88	27
Accuracy			**0.76**	497
Macro Avg	0.74	0.70	0.70	497
Weighted Avg	0.78	0.76	0.76	497

Table 5.1: Classification report on the 20% held-out test set.

The model performs remarkably well on most classes, with F1-scores above 0.90 for several professions such as *Accountant*, *Chef*, *Construction*, and *Teacher*. These results confirm the robustness of the trained classifier even in a complex, multi-class scenario.
However, a few classes show lower performance, particularly those with very limited representation in the dataset.

5.4.3 Confusion Matrix and ROC Analysis

Figure 5.6 displays the confusion matrix obtained during the final evaluation phase. This visualization provides insights into how effectively the model distinguishes between the various job categories. As seen, the classifier performs reliably across most of the 24 classes, with well-defined diagonal dominance indicating successful predictions.
That said, a significant exception concerns the **AUTOMOBILE** class, for which the model fails to correctly classify any resume in the test set. This poor performance can be explained by two key issues:

- **Severe class imbalance**: the AUTOMOBILE category is heavily underrepresented, with only 9 resumes in the entire dataset. This makes it one of the least frequent classes overall.

- **Limited test representation**: just 2 resumes from this class appear in the 20% held-out test set. Such a small sample size not only makes evaluation unreliable, but also significantly limits the model's ability to learn meaningful patterns for this category—even when SMOTE is used to artificially balance the training data.

Despite these limitations, the model exhibits strong performance on the vast majority of classes. Figure 5.7 further supports this observation by presenting the ROC curves for each individual class. Most curves display a high area under the curve (AUC), which confirms the classifier's ability to distinguish between positive and negative instances across a range of thresholds. As expected, the AUTOMOBILE class displays poor ROC characteristics due to its data scarcity.

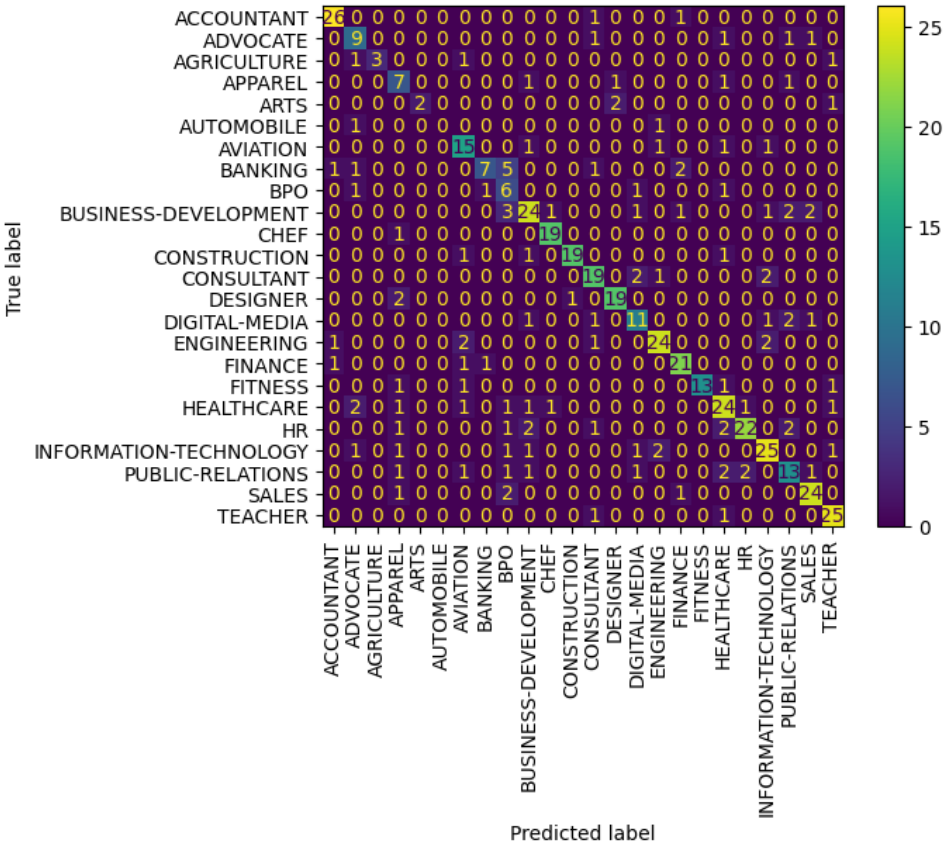


Figure 5.6: Final confusion matrix on the held-out test set.

5.4.4 Conclusion

This final evaluation step validates the robustness and generalization capabilities of the best-performing SVM model when applied to previously unseen data. The combination of SBERT-based text embeddings and SVM classification proves highly effective in tackling the challenge of multi-class resume classification, even in the presence of significant class imbalance.

The results show that the model not only achieves strong accuracy but also maintains high precision, recall, and F1-scores for the majority of classes. In particular, professions such as *Accountant*, *Chef*, *Construction*, and *Teacher* are classified with remarkable reliability.

Nonetheless, the AUTOMOBILE class clearly illustrates the limitations imposed by extremely sparse data. With only a handful of training and test examples available, the model is unable to capture representative features for this class, despite the use of SMOTE to synthetically augment the minority class during training.

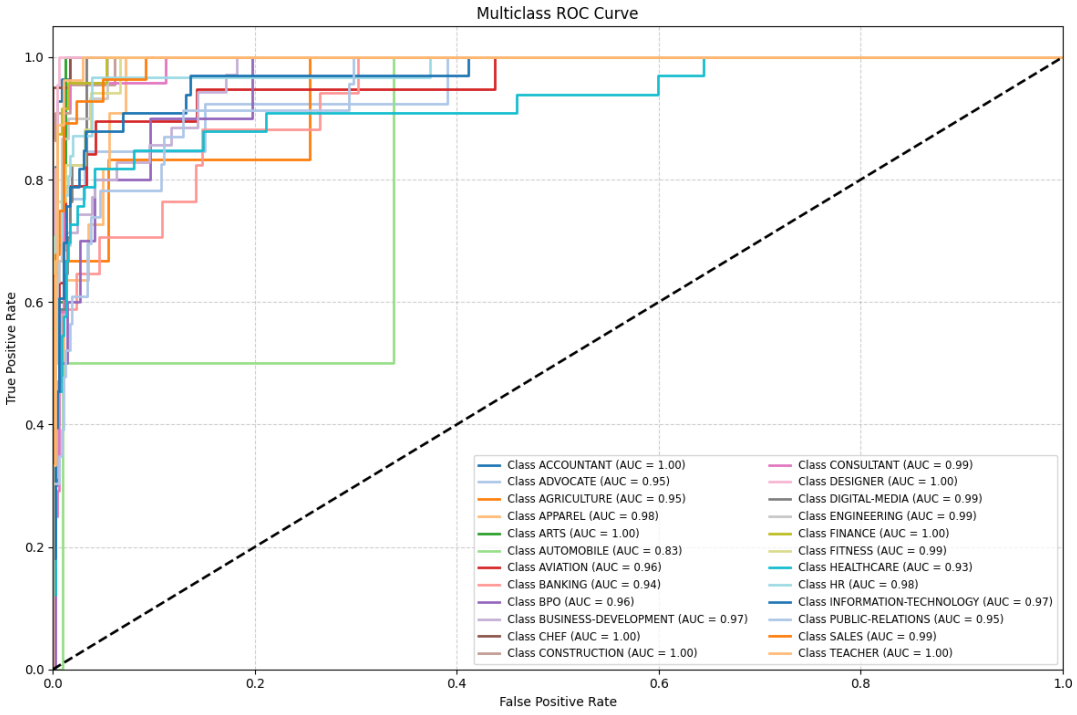


Figure 5.7: Class-wise ROC curves for the final model.

6 Model Explainability

After identifying the best-performing model—a Support Vector Machine (SVM) trained on SBERT sentence embeddings—we aimed to explore the interpretability of its predictions. The goal was to understand which parts of the input text influenced the model's classification decisions the most for each job category.

However, interpretability in models based on **pre-trained embeddings** like SBERT poses significant challenges. SBERT transforms entire sentences into dense, high-dimensional vectors that capture semantic meaning, without maintaining explicit links to individual words. Unlike models based on bag-of-words or TF-IDF, where each feature directly corresponds to a word or n-gram, SBERT's embeddings represent abstract sentence-level semantics. This makes it inherently difficult to trace a prediction back to specific words or tokens in the original input.

6.1 SHAP: SHapley Additive exPlanations

To approximate which textual components influenced the model's predictions, we used SHAP (SHapley Additive exPlanations), a model-agnostic interpretability method rooted in cooperative game theory. SHAP assigns an importance score to each feature based on its contribution to the model's output.

In our case, we applied SHAP to the SVM classifier using the `shap.plots.text()` visualization method. This approach highlights individual words or phrases in the original text, coloured according to their positive or negative contribution to the predicted class. Rather than displaying a ranked bar chart of word importances, this format provides an intuitive, in-text explanation by directly overlaying the attributions on the input text. An example of such a visualization is shown in Figure 6.1.

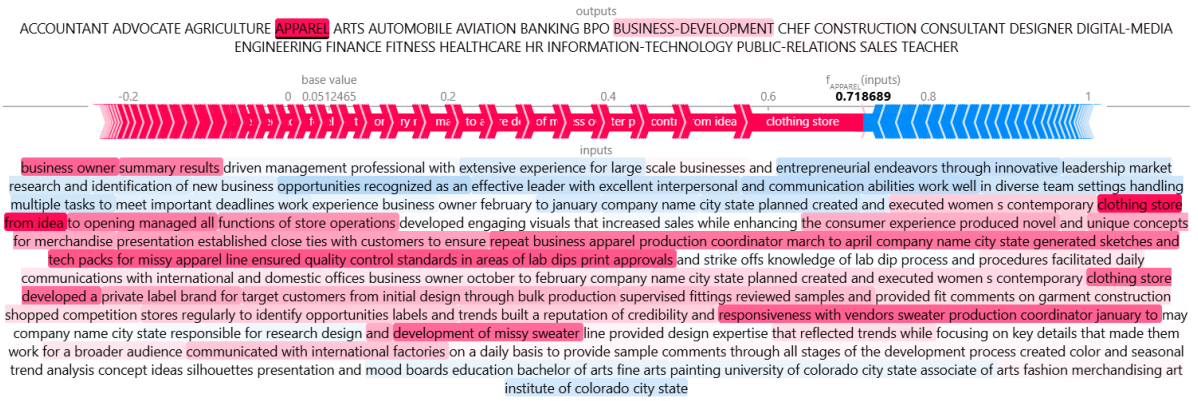


Figure 6.1: Example SHAP text explanation

6.2 Observations

Despite the challenges posed by SBERT's opaque embedding space, the SHAP visualizations provided a useful approximation of which parts of the text influenced the model's predictions. We observed that, in many cases, there was a clear correlation between the most highlighted words or phrases and the target job class. For example, words related to technical skills or industry-specific terminology were often marked as important by SHAP when classifying resumes into corresponding job categories.

However, there are important limitations. Since SHAP cannot access SBERT's internal tokenization, it segments the text using its own method, which may not align with the way SBERT processes inputs.

This discrepancy occasionally results in explanations that include function words (such as articles or conjunctions) or merge/split phrases in ways that do not reflect the true semantic parsing performed by SBERT.
Nonetheless, the interpretability offered by SHAP—while imperfect—still provides valuable insight into the model's behaviour. It helps validate that the classifier is generally attending to meaningful portions of the text, even if the exact attributions are approximated.

6.3 Conclusion and Limitations

Explaining predictions made by models built on SBERT embeddings is inherently difficult due to the dense, abstract nature of the embeddings and the lack of direct correspondence between input tokens and features. This makes traditional feature-based interpretation methods less straightforward to apply.
However, using SHAP with text-based visualization techniques allows us to approximate which words or phrases contribute most to a classification decision. While the segmentation used by SHAP does not match SBERT's internal tokenization, the visualizations still reveal a consistent alignment between important text segments and the predicted classes.
These insights can assist with validating model behaviour and provide a foundation for future development of more transparent or intrinsically interpretable models. Further work could explore explainability techniques better aligned with transformer architectures or utilize attention weights directly, where applicable.

7 Resume & Job Description Matching System

Another feature of the project is the **Job Description & Resume Matching System**, a module designed to automate the process of identifying the most relevant resumes (CVs) for a given job description. By applying advanced Natural Language Processing (NLP) techniques, the system retrieves and ranks the top 5 candidate resumes that best match a job posting, significantly assisting recruiters in the candidate selection process.

We use SBERT (Sentence-BERT) to generate semantically meaningful vector representations of both job descriptions and resumes. We have already analyzed this embedding system in Section 4.4.

7.1 Cosine Similarity

Once the embedding vectors are obtained, the similarity between a job description and each resume is computed using **cosine similarity**.
Cosine similarity measures the cosine of the angle between two vectors, indicating how similar their directions are, regardless of their magnitude. It is defined as:

$$\text{cosine\_similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} \tag{7.1}$$

Where:

- A and B are the embedding vectors of the job description and the resume;
- $A \cdot B$ is the dot product of the two vectors;
- $\|A\|$ and $\|B\|$ are the Euclidean norms (magnitudes) of the vectors.

The resulting similarity score ranges between:

- 1: perfect similarity (same direction),
- 0: no similarity (orthogonal vectors),
- -1: completely opposite directions (rare in semantic text).

7.2 Matching and Output

Given a specific job description, the system performs the following steps:

1. Compute the embedding vector of the job description using SBERT;
2. Compute the cosine similarity between this vector and the embeddings of all available resumes;
3. Sort the resumes by similarity score in descending order;
4. Select the top 5 resumes as the best matches.

```
similarity_vector = cosine_similarity(job_embed, resumes_embed).flatten()

top_n = 5
top_matches = similarity_vector.argsort()[::-1][:top_n]

results = []
for rank, cv_idx in enumerate(top_matches, start=1):
```

```
    results.append({
        'rank':             rank,
        'job_id':           job_sample['job_id'],
        'job_title':        job_sample['title'],
        'cv_id':            df_resumes.iloc[cv_idx]['ID'],
        'cv_category':      df_resumes.iloc[cv_idx]['Category'],
        'similarity_score': similarity_vector[cv_idx]
    })
```

7.3 Evaluation Cosine similarity

A critical challenge in evaluating the performance of the Job Description & Resume Matching System was the lack of a labelled *ground truth*. In this context, ground truth refers to a dataset where each job description is explicitly annotated with the most relevant resumes, serving as a benchmark to assess the accuracy of a matching algorithm. However, since no such labelled dataset was available, an alternative evaluation strategy was required.

To address this, we leveraged a **Large Language Model (LLM)** to simulate human judgment in assessing the relevance between resumes and job descriptions. Specifically, we employed **`gemini-2.0-flash`**, a high-performance LLM, to assign relevance scores to candidate-job pairs. The model was prompted to evaluate whether the candidate's professional domain aligns with that of the job description and to provide a numerical score from 0 to 10, where 0 indicates no match and 10 indicates a perfect match.

The prompt used for this evaluation is shown below:

```
prompt = f"""Analyze the match between this RESUME and JOB DESCRIPTION.
As a Human Resources recruiter, your task is to evaluate whether the professional field of the candidate aligns with
↪  that of the job description.

RESUME:
{resume_text}

JOB DESCRIPTION:
{job_description}

Given a resume and a job description, assign an integer score from 0 to 10 indicating how well the resume matches
↪  the position.

Provide only a single integer score representing how well this resume matches the job requirements.
Don't provide any explanation, just return the integer score between 0 and 10.
"""
```

The prompt was submitted to the LLM using the following API call:

```
response = client.models.generate_content(
    model = "gemini-2.0-flash",
    contents = prompt,
)
```

Due to usage limitations of the Gemini API, we evaluated a subset of over 3,000 unique job description and resume pairs. For each pair, we computed two scores: the cosine similarity between their SBERT embeddings, and the corresponding evaluation score assigned by Gemini based on the prompt described previously.

To assess the alignment between our embedding-based similarity measure and the LLM-based evaluation, we conducted a quantitative correlation analysis. The goal was to understand to what extent the cosine similarity reflects the same matching judgment that Gemini provides.

The evaluation metrics are summarized below:

- **Mean Absolute Error (MAE)**: 1.541
- **Mean Squared Error (MSE)**: 4.146
- **Pearson Correlation Coefficient**: 0.623
- **Percentage of predictions within ±2 points**: 75.9%

These results indicate a moderately strong positive correlation between cosine similarity and Gemini evaluations. In particular, the Pearson correlation coefficient of 0.623 suggests that, although the embedding-based similarity metric does not fully replicate human-level assessment, it captures a significant portion of the semantic alignment perceived by the LLM. Furthermore, the fact that over 75% of predictions fall within a ±2 score margin demonstrates a reasonable level of reliability for practical use.
To further understand the relationship between the embedding-based similarity scores and the Gemini-evaluated scores, we generated a two-dimensional bubble plot, shown in Figure 7.1. This visualization captures the joint distribution of cosine similarity scores (x-axis) and Gemini scores (y-axis), along with the number of occurrences for each score pair.

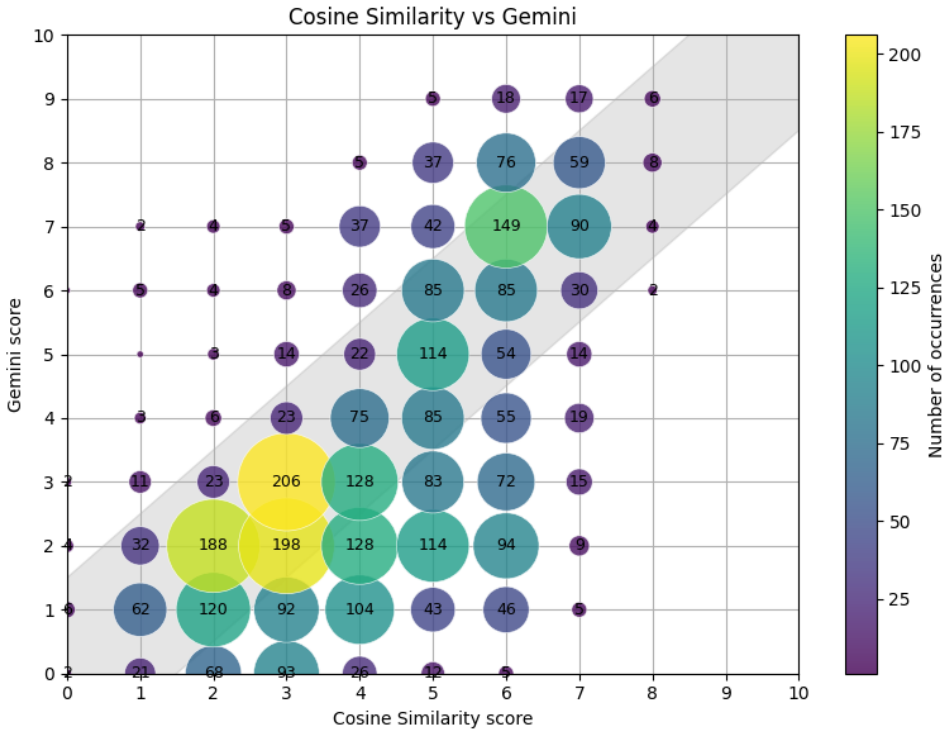


Figure 7.1: Joint distribution of Cosine Similarity and Gemini scores

Each circle in the plot represents a specific pair of cosine similarity score (horizontal axis) and Gemini score (vertical axis). The size and colour of the circle correspond to the frequency of that score pair, as indicated by the colour bar on the right-hand side.
A dense concentration of data points within the diagonal margin suggests that the cosine similarity score often aligns closely with the Gemini score, further reinforcing the consistency observed in the quantitative metrics. This visual evidence supports the conclusion that cosine similarity provides a reasonable semantic approximation for matching relevance, even in the absence of labelled supervision.
Overall, this combined evaluation validates the use of cosine similarity as a viable method for job-resume matching, especially when labeled data is not available.

8 Graphical User Interface

The user interface (UI) of the project has been developed to be simple, intuitive, and accessible. It is implemented using HTML, CSS, and JavaScript for the frontend, and Python with the Flask framework for the backend. The interface provides two main functionalities: matching resumes to job descriptions and classifying resumes into job categories.

8.1 Job Description and Resume Matching Page

As shown in Figure 8.1, this is the first interface the user encounters. It is specifically designed for companies, recruiters, or employment agencies who wish to identify suitable candidates for a specific job offering.
In this section, the user can input the text of a job description in a dedicated field. The system then compares the job posting with all resumes stored in the database and returns the top 5 most relevant candidates. This similarity is computed using the SBERT-based embedding model that captures the semantic alignment between job requirements and candidate profiles.

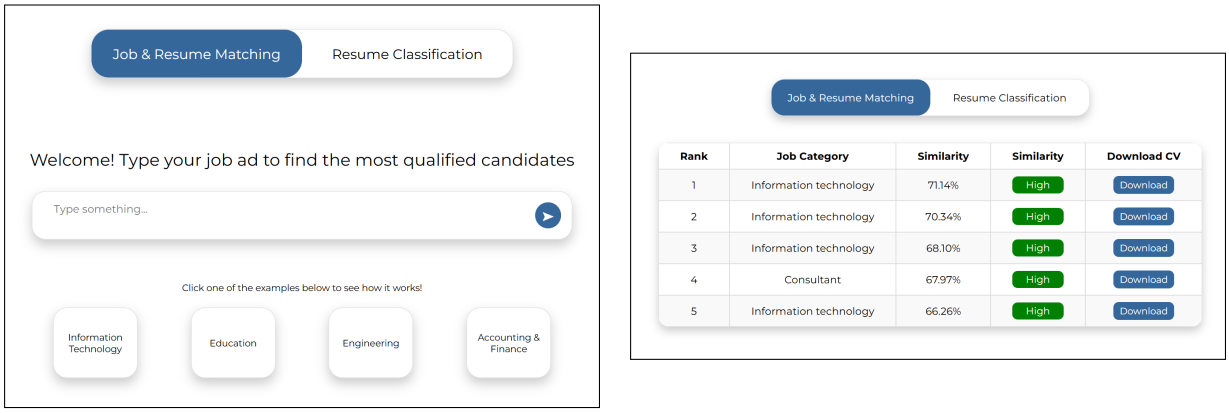


Figure 8.1: UI Job & Resume Matching

Example use case: Suppose an employment agency receives a request from a company looking to hire a mechanical engineer. By inserting the job description into the system, the agency can automatically retrieve the best matching resumes from their archive—saving time, reducing manual screening, and increasing candidate matching quality.

8.2 Resume Classification Page

The second main functionality is the resume classification, visually depicted in Figure 8.2.
This interface is designed primarily for job seekers. Users can either paste the textual content of their resume or upload it in PDF format. Once submitted, the backend processes the document and predicts the top 3 most relevant job categories from a fixed set of 24 classes, providing a quick summary of how their resume is interpreted by the machine learning model.
This tool offers candidates valuable feedback on how their experience and skills align with various professional roles, potentially helping them refine their resume or explore new career paths.

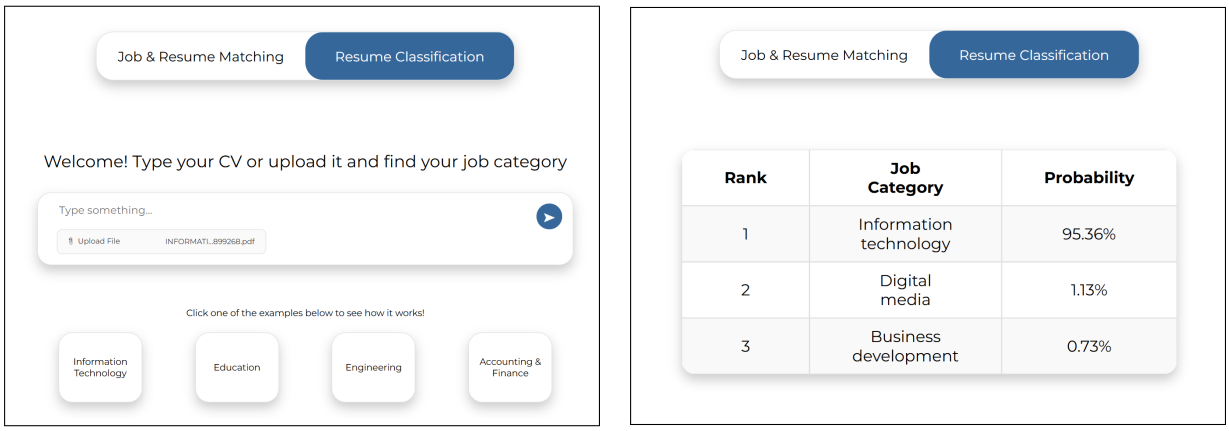


Figure 8.2: UI Resume Classification

Bibliography

[1] Ahmed Heakl, Youssef Mohamed, Noran Mohamed, Aly Elsharkawy, and Ahmed Zaky. Resumeatlas: Revisiting resume classification with large-scale datasets and large language models, 2024.

[2] Tejaswini K., V. Umadevi, Shashank M. Kadiwal, and Sanjay Revanna. Design and development of machine learning based resume ranking system. *Global Transitions Proceedings*, 2022.

[3] R. Pal, S. Shaikh, S. Satpute, and S. Bhagwat. Resume classification using various machine learning algorithms. In *ITM Web of Conferences, Proceedings of ICACC 2022*. EDP Sciences, 2022.

[4] B. Surendiran, T. Paturu, H. V. Chirumamilla, and M. N. R. Reddy. Resume classification using ml techniques. In *2023 International Conference on Signal Processing, Computation, Electronics, Power and Telecommunication (IConSCEPT)*. IEEE, 2023.