ResumeIQ – AI-Based Resume Analysis Platform

Project Report submitted in partial fulfillment of
The requirements for the degree of

MASTER OF COMPUTER APPLICATIONS

Of

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

By

MD SADRUDDIN ANSARY, Roll no. 29171024002
SOUMYAJIT SARKAR, Roll no. 29171024019
BODHISATTWA BASU, Roll no. 291710240
ANIRBAN ROUTH, Roll no. 29171024023

Under the guidance of

DEBASHIS CHOWDHURY

DEPARTMENT OF COMPUTER APPLICATIONS

NETAJI SUBHASH ENGINEERING COLLEGE
TECHNO CITY, GARIA, KOLKATA – 700 152

Academic year of pass out 2025-2026

CERTIFICATE

This is to certify that this project report titled "**ResumeIQ – AI-Based Resume Analysis Platform**" submitted in partial fulfillment of requirements for award of the degree Master of Computer Application of Maulana Abul Kalam Azad University of Technology is a faithful record of the original work carried out by,

MD SADRUDDIN ANSARY, **Roll no.** 29171024002 , **Regd. No.** 242910510003 of 2024-2025
SOUMYAJIT SARKAR, **Roll no.** 29171024019, **Regd. No.** 242910510026 of 2024-2025
BODHISATTWA BASU, **Roll no.** 29171024012, **Regd. No.** 242910510008 of 2024-2025
ANIRBAN ROUTH, **Roll no.** 29171024023, **Regd. No.** 242910510006 of 2024-2025
under my guidance and supervision.

It is further certified that it contains no material, which to a substantial extent has been submitted for the award of any degree/diploma in any institute or has been published in any form, except the assistances drawn from other sources, for which due acknowledgement has been made.

Date:……..

_____
Guide's signature

DEBASHIS CHOWDHURY

Sd/_____

Head of the Department

Master of Computer Application

NETAJI SUBHASH ENGINEERING COLLEGE
TECHNO CITY, GARIA, KOLKATA – 700 152

DECLARATION

We hereby declare that this project report titled

“ResumeIQ – AI-Based Resume Analysis Platform”

is our own original work carried out as a under graduate student in Netaji Subhash Engineering College except to the extent that assistances from other sources are duly acknowledged.

All sources used for this project report have been fully and properly cited. It contains no material which to a substantial extent has been submitted for the award of any degree/diploma in any institute or has been published in any form, except where due acknowledgement is made.

Student’s names:	Signatures:	Dates:
……..	……..	……
……..	……..	……
……..	……..	……
……..	……..	……

CERTIFICATE OF APPROVAL

We hereby approve this dissertation titled

“ResumeIQ – AI-Based Resume Analysis Platform”

carried out by

MD SADRUDDIN ANSARY, **Roll no.** 29171024002 , **Regd. No.** 242910510003 of 2024-2025

SOUMYAJIT SARKAR, **Roll no.** 29171024019, **Regd. No.** 242910510026 of 2024-2025

BODHISATTWA BASU, **Roll no.** 29171024012, **Regd. No. 242910510008** of 2024-2025

ANIRBAN ROUTH, **Roll no.** 29171024023, **Regd. No.** 242910510006 of 2024-2025

under the guidance of

DEBASHIS CHOWDHURY

of Netaji Subhash Engineering College, Kolkata in partial fulfillment of requirements for award of the Master of Computer Application of Maulana Abul Kalam Azad University of Technology.

Date:……..

Examiners’ signatures:

1. …….
2. …….
3. …….

Acknowledgement

We would like to express our sincere gratitude to our project mentor **Mr. Debashis Chowdhury** for his invaluable guidance, constant encouragement, and insightful suggestions throughout the development of this project. His expertise, constructive feedback, and continuous support played a crucial role in shaping the successful completion of our work.

We are also thankful to our department and institution for providing us with the necessary resources, infrastructure, and learning environment that enabled us to carry out this project effectively.

We would like to extend our appreciation to our faculty members and peers for their cooperation, motivation, and support during the course of this project.

Finally, we are grateful to our families and friends for their patience, encouragement, and moral support, which helped us stay motivated and focused throughout this endeavor.

Md Sadruddin Ansary

Soumyajit Sarkar

Bodhisattwa Basu

Anirban Routh

Dated:……

ABSTRACT

The rapid growth of online recruitment platforms has resulted in organizations receiving extremely large numbers of resumes for each job opening, making manual screening slow, inconsistent, and inefficient. Existing Applicant Tracking Systems (ATS) attempt to address this issue but rely heavily on rigid keyword matching and rule-based filtering, which often fail to capture the true skills and experience of candidates. As a result, many qualified applicants are overlooked due to formatting variations, synonym usage, or differences in writing style.

This project presents the design and development of an intelligent, AI-based resume screening and ATS scoring system that combines machine learning and natural language processing techniques. The proposed system classifies resumes into job categories, constructs role-specific prototype profiles representing ideal candidates, and evaluates incoming resumes using semantic similarity, keyword relevance, and an AI-based scoring model. The system supports multiple resume formats and includes a fallback mechanism to ensure reliable scoring even with incomplete or noisy data.

Experimental results demonstrate that the proposed approach achieves high screening accuracy, exceeding 97%, while offering transparent and interpretable scoring. The system improves fairness, reduces bias, and significantly enhances the efficiency of recruitment processes, making it suitable for practical deployment in modern HR environments.
Our experimental predictions show that using the given algorithms has yielded more than 97% accuracy. These results suggest that NLP and ML techniques in this study could be used for developing an efficient ATS.

Key words: Artificial Intelligence (AI), Resume Screening, Applicant Tracking System (ATS), Machine Learning (ML), Natural Language Processing (NLP), Semantic Analysis, Prototype-Based Modeling, Candidate Evaluation

CONTENTS

Chapter 1
Introduction

The recruitment landscape has undergone a profound transformation over the last two decades. Traditional hiring processes, which relied heavily on in-person applications, interviews, and paper-based resume submissions, have gradually shifted to fully digital ecosystems. The advent of online job portals, professional networking platforms, and global talent marketplaces has made it possible for organizations to reach an unprecedented number of applicants for every open position. While this evolution has democratized access to employment opportunities, it has also introduced significant operational challenges for human resource (HR) departments. Modern organizations often receive **hundreds or even thousands of digital resumes per vacancy**, creating an overwhelming workload for HR personnel tasked with identifying the most suitable candidates. This volume renders **manual evaluation impractical**, expensive, and prone to human error.

Human recruiters, while highly skilled, are limited by time and cognitive bandwidth. Evaluating large numbers of resumes not only slows the hiring process but also increases the likelihood of oversight and inconsistent assessments. Subtle differences in wording, variations in resume structure, and formatting inconsistencies can lead to the unintentional rejection of highly qualified candidates. These inefficiencies have prompted organizations to adopt **Applicant Tracking Systems (ATS)**—software solutions designed to automatically filter, rank, and manage incoming resumes. ATS platforms are now a ubiquitous component of enterprise recruitment strategies, particularly in large corporations and high-volume hiring scenarios.

Challenges in Modern Recruitment

Despite widespread adoption, traditional Applicant Tracking Systems (ATS) have significant limitations. Most systems operate primarily on keyword-based matching and rigid, rule-based filtering, which makes them highly sensitive to the exact choice of words, abbreviations, and formatting styles used in resumes. For example, a candidate who lists **“Software Engineer”** in one resume and **“Software Developer”** in

another might be treated differently by an ATS that only scans for exact keyword matches, even though the roles are functionally similar in most organizations. Likewise, industry-specific variations—such as “ML Engineer” vs. “Machine Learning Specialist” or “Front-End Developer” vs. “UI Developer”—may not be recognized as synonymous unless manually configured. This lack of linguistic flexibility reduces the system’s ability to generalize effectively and often penalizes applicants for trivial differences in phrasing.

Formatting issues introduce another layer of inaccuracy. Resumes submitted as PDF files may be partially unreadable if the ATS struggles to parse non-standard fonts, multi-column layouts, tables, embedded icons, or graphical skill indicators. Many candidates use visually enhanced templates that include charts, progress bars, or decorative elements, which traditional systems interpret incorrectly or fail to process altogether. As a result, substantial portions of a resume—sometimes entire sections such as skills, experience, or certifications—can be omitted during parsing. This rigid and error-prone extraction pipeline frequently leads to **false negatives**, where highly qualified candidates are filtered out simply because their resumes do not conform to the system’s preferred structure.

A deeper challenge lies in the **semantic limitations** of traditional ATS. While keyword frequency can signal the presence of a term, it cannot capture the subtlety, relevance, or depth of the competency being described. For instance, a candidate who writes, “Completed multiple machine learning projects involving regression, classification, and deep neural networks,” clearly demonstrates practical experience. In contrast, a resume that includes the isolated keyword “machine learning” within a course list or a generic objective statement provides little insight into actual skill level. Conventional ATS are incapable of distinguishing between these contexts, leading to **shallow evaluations** that do not reflect a candidate’s true qualifications.

Additionally, keyword-based models fail to recognize the relationships between skills, technologies, and job responsibilities. They cannot infer that someone experienced in “TensorFlow” or “PyTorch” is likely to have hands-on exposure to machine learning principles, or that familiarity with “REST APIs” and “microservices” may imply backend development experience. This absence of semantic understanding makes

traditional systems brittle and heavily dependent on manual configuration by HR teams, who must continuously update keyword lists for every role. As job descriptions evolve or new technologies emerge, these lists may become outdated, further degrading system accuracy and increasing the risk of both **false positives** and **false negatives**.

Ultimately, the rigidity of traditional ATS undermines their objective of efficient and fair candidate evaluation. By ignoring context, meaning, and nuanced experience, they often reward candidates who strategically optimize their resumes for keyword density rather than those who genuinely possess the required skills. This gap between surface-level matching and true competency underscores the urgent need for intelligent, context-aware, and semantically rich resume screening solutions.

Objectives of the Present Work

Modern recruitment has become an increasingly digitized, global, and data-intensive process. With the widespread adoption of online job portals, professional networking platforms, and automated application systems, organizations today routinely receive hundreds or even thousands of resumes for a single job opening. While this digital transformation has broadened access to employment opportunities and improved reach for employers, it has simultaneously introduced significant operational challenges for human resource (HR) departments. The primary objective of the present work is to design and develop an intelligent, AI-driven Applicant Tracking System (ATS) capable of evaluating candidate resumes accurately, fairly, and transparently at scale. This objective directly addresses the growing disconnect between the simplistic mechanisms employed by traditional ATS tools and the semantically rich, diverse, and unstructured nature of modern resumes.

A core objective of this research is to substantially improve the accuracy of automated resume screening. Conventional ATS platforms rely heavily on keyword-based matching and rigid rule-based filters. Such approaches primarily evaluate resumes based on surface-level textual overlap, often failing to capture the true relevance, depth, and applicability of a candidate's experience. Candidates may describe equivalent skills using different terminology, narrative styles, or contextual framing, leading to inconsistent and unreliable screening outcomes. The present work aims to overcome these shortcomings by introducing semantic similarity-based evaluation, enabling resumes to be assessed according to contextual meaning and alignment with job roles rather than exact keyword correspondence.

Another fundamental objective of this research is to **enhance fairness and inclusivity within automated recruitment processes**. Traditional ATS systems unintentionally favor candidates who are familiar with ATS-optimized resume writing practices, such as standardized templates, keyword-heavy phrasing, and rigid formatting conventions. Applicants from diverse educational, linguistic, or cultural backgrounds may articulate their competencies differently, resulting in lower ATS scores despite possessing equivalent qualifications. This research seeks to minimize such systemic bias by emphasizing semantic relevance, contextual interpretation, and role alignment, thereby promoting

equitable candidate evaluation and supporting diversity in hiring decisions.

Transparency and interpretability constitute a central objective of the present work. Many modern AI-driven recruitment tools employ complex deep learning architectures that function as black-box systems, providing little or no explanation for their decisions. This lack of transparency reduces trust among HR professionals, limits accountability, and complicates compliance with ethical and regulatory standards. The present research aims to develop an interpretable ATS framework in which evaluation criteria and scoring components are explicitly represented. Recruiters can therefore understand how similarity measures, relevance indicators, and predictive models contribute to final candidate rankings.

Scalability represents another major objective of this work. Large enterprises and recruitment agencies often operate under strict hiring timelines and must process vast numbers of resumes efficiently. Manual screening under such conditions is not only time-consuming but also inconsistent due to evaluator fatigue and subjective judgment. The proposed system is designed to process large volumes of resumes rapidly while maintaining analytical depth, consistency, and reliability. This balance between scalability and precision is critical for real-world deployment.

Adaptability to evolving job roles and industry requirements is an additional objective of this research. In rapidly changing fields such as information technology, data science, and digital marketing, job descriptions and skill requirements evolve continuously. Traditional ATS platforms rely on static keyword lists and manually configured rules, which quickly become outdated. **The present work aims to introduce adaptive role modeling mechanisms that evolve naturally with new data, reducing dependence on manual updates and ensuring long-term relevance.**

Robustness to noisy, incomplete, or inconsistently formatted input data is another important objective. Real-world resumes frequently contain missing sections, unconventional layouts, scanned content, or ambiguous descriptions. Conventional ATS tools often perform poorly under such conditions, leading to unreliable evaluations. The proposed system seeks to maintain meaningful assessment through flexible

evaluation strategies and fallback mechanisms, reflecting realistic deployment environments.

From a practical standpoint, this research is motivated by the need for deployable, usable, and resource-efficient systems. Many advanced AI models demonstrate high performance in experimental settings but fail to translate into practical tools due to computational cost or usability constraints. An explicit objective of this work is to ensure implementation feasibility by developing a system that can be deployed as a functional application for real-world use by HR professionals.

Furthermore, this research aims to bridge the gap between classical machine learning approaches and modern natural language processing techniques. Classical models offer interpretability and efficiency but lack semantic depth, while deep learning models provide strong contextual understanding at the cost of transparency and computational complexity. The objective of this work is to integrate the strengths of both paradigms into a hybrid system that achieves high accuracy without sacrificing interpretability or efficiency.

The overarching objective of this research is to **contribute a structured, extensible, and ethically responsible framework** for intelligent resume screening. By addressing accuracy, fairness, interpretability, scalability, adaptability, and robustness within a unified system, the present work aims to advance the state of automated recruitment technology and provide a foundation for future academic research and industrial innovation.

BACKGROUND AND MOTIVATION OF THE PROBLEM

- Recruitment and talent acquisition have undergone a significant transformation with the advancement of digital technologies. Organizations across industries increasingly rely on online platforms, job portals, and professional networks to attract candidates. While this digital transformation has expanded the talent pool, it has also introduced a major operational challenge: the exponential growth in the number of resumes received for each job opening. In many cases, a single job posting can attract

hundreds or even thousands of applications within a short span of time.

- Traditionally, resume screening has been performed manually by human recruiters or hiring managers. This process involves reading resumes, identifying relevant skills and experience, and deciding whether a candidate should move forward in the hiring pipeline. Although human judgment can be insightful, manual resume screening is inherently slow, labor-intensive, and subjective. Recruiters may unintentionally introduce bias based on factors such as resume formatting, writing style, or personal assumptions. Additionally, time pressure often forces recruiters to skim resumes, increasing the likelihood of overlooking qualified candidates.

- To address these challenges, Applicant Tracking Systems (ATS) were introduced as automated tools for managing recruitment workflows. Conventional ATS solutions primarily rely on keyword-based filtering mechanisms. These systems scan resumes for predefined keywords related to job requirements and rank candidates accordingly. While keyword-based systems improve processing speed, they suffer from several limitations. They fail to understand semantic meaning, context, and skill equivalence. For instance, a candidate proficient in “Natural Language Processing” may be overlooked if the job description emphasizes “Text Analytics,” despite the conceptual overlap.

- Moreover, modern candidates often optimize resumes specifically to pass ATS filters by stuffing keywords, sometimes at the expense of clarity or genuine expertise. This leads to inaccurate rankings where resumes with high keyword density outperform resumes with stronger real-world relevance. Such limitations reduce the

effectiveness of traditional ATS systems and raise concerns regarding fairness, accuracy, and reliability.

The emergence of Artificial Intelligence, Machine Learning, and Natural Language Processing offers a promising solution to these challenges. Machine learning models can be trained on historical resume data to identify patterns associated with specific job roles. NLP techniques enable computers to process and analyze unstructured text data, extracting meaningful features such as skills, experience, and professional context. When combined, these technologies allow automated systems to evaluate resumes in a manner that more closely resembles human understanding, but with significantly greater speed and consistency.

The motivation behind this project is to develop an intelligent resume screening system that overcomes the shortcomings of traditional ATS solutions. The system aims to automatically extract text from resumes, preprocess and normalize the content, classify resumes into appropriate job categories using machine learning, and compute a quantitative compatibility score that reflects the alignment between a resume and the expected role requirements.
Another key motivation is transparency and usability. Many commercial ATS platforms operate as black-box systems, providing little insight into how scores are generated. This project emphasizes interpretability by exposing intermediate metrics such as content similarity and keyword overlap, allowing users to understand why a particular score was assigned. This makes the system suitable not only for recruiters but also for job seekers who wish to improve their resumes.

Related Earlier Work

The evolution of automated recruitment technologies provides the necessary context for the present study. The trajectory of this field has been defined by a continuous effort to structure the unstructured nature of human language into actionable data for decision-makers.

The Era of Digitization and Keyword Parsing The genesis of the Applicant Tracking System (ATS) dates back to the late 1990s and early 2000s, coinciding with the rapid expansion of the internet. Early research and industrial solutions focused primarily on the digitization of the recruitment workflow. Systems developed during this period were essentially database management tools designed to replace physical filing cabinets. The primary mechanism for screening was **Rule-Based Parsing**. Researchers established frameworks based on Boolean logic, allowing recruiters to filter large databases using specific search strings (e.g., "AND", "OR", "NOT"). These foundational works were pivotal in establishing the viability of digital recruitment, proving that large volumes of applicant data could be centralized and queried. They successfully solved the problem of storage and retrieval, laying the groundwork for the analytical systems that would follow.

The Advent of Statistical Machine Learning As computational capabilities expanded in the 2010s, the academic focus shifted from simple retrieval to predictive classification. This era marked the introduction of **Classical Machine Learning (ML)** into recruitment. A significant body of literature emerged exploring the application of statistical classifiers—such as **Naïve Bayes**, **Support Vector Machines (SVM)**, and **Decision Trees**—to the task of resume categorization.

Researchers in this phase concentrated heavily on **Feature Engineering**. Studies demonstrated how textual data could be converted into numerical vectors using **Bag-of-Words (BoW)** and **Term Frequency-Inverse Document Frequency (TF-IDF)** models. By identifying statistical patterns in word usage, these systems could automatically route resumes to the correct department (e.g., distinguishing a "Marketing" resume from a "Finance" resume) with a high degree of reliability. This body of work was instrumental in proving that automated systems could go beyond exact keyword matching and begin to identify genre and category based on probabilistic word distributions.

Deep Learning and Semantic Context In recent years, the literature has expanded to include **Deep Learning** and **Natural Language Processing (NLP)**. The introduction of Word Embeddings (such as

Word2Vec and GloVe) represented a major leap forward. Unlike previous statistical methods that treated words as independent symbols, embedding-based approaches mapped words to high-dimensional vectors where distance represented semantic similarity. This allowed systems to recognize that "programmer" and "developer" are related concepts. Most recently, transformer-based architectures (like BERT) have been applied to extract specific entities (names, skills, dates) from resumes with remarkable precision. These modern contributions have advanced the field significantly, enabling systems to handle the linguistic nuances and synonymous terms that characterize human communication.

The present work builds directly upon this rich lineage. It integrates the efficiency of classical vectorization established in the ML era with the semantic goals of the Deep Learning era, aiming to synthesize these approaches into a tool that is both computationally lightweight and semantically aware.

Innovation and Contribution

While the existing body of literature provides well-established techniques for resume classification, information extraction, and keyword-based matching, most prior approaches remain limited to coarse-grained categorization or opaque scoring mechanisms. The present work differentiates itself by moving beyond simple resume labeling and instead proposing a practically oriented, interpretable, and adaptive screening architecture tailored to the operational realities of modern Human Resource (HR) environments.

The core innovation of this research lies not in introducing entirely new machine learning algorithms, but in how existing, well-understood techniques are systematically integrated to address three persistent challenges in automated recruitment systems: **interpretability**, **adaptive baselining**, and **granular suitability scoring**. Rather than treating resume screening as a single monolithic prediction task, the proposed framework decomposes the problem into meaningful,

explainable stages that align closely with how human recruiters reason about candidate suitability.

The novel contributions of this work are structured around three complementary pillars:

1. **Prototype-Based Role Modeling** for data-driven baselining.
2. **A Hybrid Classification–Regression Architecture** for decoupled decision-making.
3. **The implementation of transparent, "glass-box" evaluation metrics** for explainable screening.

Prototype-Based Role Modeling (Data-Driven Baselining)

The primary conceptual contribution of this research is the introduction of **Prototype-Based Role Modeling** as a mechanism for resume evaluation. Traditional supervised learning approaches in the recruitment domain typically frame resume screening as a discrete classification task, where each resume is mapped directly to a predefined role label (e.g., "Software Engineer" or "Data Scientist"). While such classification is useful for organizing applications, it provides no quantitative indication of how well a candidate aligns with the expectations of a role.

The present work extends this paradigm by introducing a continuous, data-driven baseline against which candidate resumes can be evaluated. Instead of relying on static job descriptions or manually defined skill lists, the system constructs a **"Master Profile"** (or prototype) for each job category directly from historical resume data used during training.

Collective Intelligence and Data-Driven Definitions

At the core of this approach is the concept of *collective intelligence*. Rather than encoding role requirements through human-authored job descriptions—which may be subjective, incomplete, or quickly outdated—the system aggregates the textual content of all resumes associated with a given role label in the training dataset. As

implemented in the training pipeline, resumes belonging to the same category are combined to form a unified textual representation, which is then transformed into a vector space using **TF-IDF**.

In this high-dimensional vector space, the aggregated representation functions as a **centroid vector**, mathematically representing the average characteristics of successful candidates for that role. Individual idiosyncrasies—such as resume formatting choices, personal hobbies, or stylistic variations—are naturally attenuated through aggregation, while recurring, role-relevant concepts (e.g., “Python,” “Machine Learning,” or “SQL” in data science roles) become more prominent. As a result, the prototype serves as a data-derived definition of an idealized role profile, grounded entirely in empirical evidence rather than manual assumptions.

Adaptive and Self-Updating Benchmarks

A definitive advantage of the proposed prototype-based strategy is its inherent adaptability and capacity for continuous measurement. Traditional Applicant Tracking Systems (ATS) typically operate on binary logic: a candidate is either a "match" or a "mismatch" based on the presence of specific boolean triggers. This binary classification is fundamentally reductive; it collapses the complex, multi-dimensional spectrum of human competence into a zero-sum decision. In contrast, the present work utilizes **Cosine Similarity** to evaluate incoming resumes not as discrete categorical matches, but as vectors situated at varying distances from an ideal center. This allows the system to quantify alignment on a continuous scale (0.0 to 1.0), supporting nuanced judgments that can distinguish between a "marginal fit" and an "exceptional fit" with high granularity.

The Geometry of Evolving Competence

From a geometric perspective, the "Master Profile" or prototype acts as a centroid—the center of gravity for a specific job cluster in high-dimensional space. This centroid is not static; it is a dynamic mathematical aggregate derived directly from the training corpus. As the recruitment landscape shifts, so too does the composition of this centroid.

Consider the rapidly evolving field of Data Science as a case study. Five years ago, the vector representation of a "Data Scientist" would have been heavily weighted towards terms like *Linear Regression*, *SVM*, and *Hadoop*. However, modern resumes in this domain increasingly feature terms such as *Deep Learning*, *MLOps*, and *Kubernetes*.

In a traditional rule-based ATS, this shift would represent a catastrophic point of failure. HR administrators would be forced to manually intervene, updating static keyword dictionaries to include "MLOps" or "CI/CD" while deprecating older terms to prevent false negatives. This process is slow, error-prone, and perpetually reactive.

Mechanism of Self-Correction

The proposed system eliminates this operational bottleneck through data-driven self-correction. When new resumes containing emerging technologies are verified (e.g., by human hiring decisions) and incorporated into the training dataset, the prototype recalculates. The new terminology naturally pulls the centroid vector toward the current state of the industry.

- **Implicit Learning:** The system learns the association between "Data Scientist" and "Deep Learning" implicitly, simply because successful candidates are now using that terminology.
- **Automatic Deprecation:** Conversely, as obsolete technologies fade from the resumes of top talent, their weight in the centroid vector diminishes.

This capability ensures that the system serves as a lag-free reflection of the real world. It captures the **realistic standard** of a job role as defined by the labor market itself, rather than the **rigid standard** defined by a potentially outdated job description. By anchoring evaluation in collective intelligence rather than manual rules, the system achieves a level of robustness and longevity that static models cannot match, effectively future-proofing the screening process against the inevitability of technological change.

Hybrid Classification–Regression Architecture

From a system design perspective, the present work introduces a **two-stage hybrid architecture** that separates domain identification from suitability scoring. Many existing ATS solutions rely on a single monolithic model that simultaneously attempts to classify resumes and determine candidate quality. Such designs often suffer from limited interpretability and reduced flexibility. In contrast, the proposed system explicitly decouples these objectives, allowing each stage to be optimized independently while maintaining coherence in the overall pipeline.

Stage 1: Domain Identification via Instance-Based Learning

The first stage of the pipeline focuses exclusively on domain identification, implemented using a **K-Nearest Neighbors (KNN)** classifier trained on TF-IDF representations of resumes. KNN was deliberately chosen for its instance-based and non-parametric nature. Unlike linear or parametric classifiers that assume predefined decision boundaries, KNN determines class membership based on local similarity within the feature space.

This property is particularly advantageous for resume data, which is inherently unstructured, noisy, and heterogeneous. Different candidates may describe similar experiences using varied terminology or formatting styles, leading to irregular and overlapping clusters in the vector space. By relying on proximity rather than rigid boundaries, KNN effectively accommodates such non-linear distributions and provides robust domain predictions without requiring complex model assumptions.

Stage 2: Suitability Scoring via Ensemble Regression

Once the resume's domain has been identified, the second stage evaluates candidate suitability using a **Gradient Boosting Regressor**. Unlike text-based classifiers, this regressor does not operate on raw

textual input. Instead, it receives high-level, semantically meaningful features derived from the relationship between the candidate resume and the corresponding prototype.

Specifically, the model uses cosine similarity to capture **semantic alignment** and a keyword overlap metric to capture **lexical precision**. This feature abstraction step ensures that the regressor focuses on role relevance rather than individual word frequencies, reducing susceptibility to noise and overfitting. Gradient boosting is employed for its ability to model non-linear relationships between these features while maintaining relatively strong interpretability compared to deep neural architectures. By separating classification and scoring, the architecture provides modularity and flexibility; improvements to one stage can be made without retraining the entire system.

Transparent "Glass-Box" Metrics

Perhaps the most practically significant contribution of this work is its strong emphasis on **explainability and transparency**. In high-stakes domains such as recruitment, opaque decision-making systems raise ethical, legal, and organizational concerns. Many deep learning–based ATS solutions produce a final score or recommendation without offering insight into how that decision was reached.

The proposed system is explicitly designed as a **"glass-box" model**, where each component of the final score is observable, interpretable, and meaningful to human decision-makers. Rather than providing a single, unexplained output, the system generates a decomposable evaluation consisting of complementary metrics.

Decomposed Evaluation Metrics

The system calculates and presents:

- **Content Match:** Derived from TF-IDF–based cosine similarity, reflecting the overall *semantic alignment* between the candidate's resume and the role prototype.
- **Keyword Overlap:** Computed through token-level intersection, highlighting the presence or absence of critical, role-specific *technical terms*.

These metrics are displayed independently through the user interface, allowing recruiters to understand *why* a candidate received a particular score. For instance, a resume may demonstrate strong conceptual alignment with a role while lacking specific tool mentions, or vice versa. This level of transparency enables informed human judgment rather than blind reliance on algorithmic output.

Decision Support Rather Than Decision Replacement

By exposing intermediate reasoning signals, the system functions as a **decision support tool** rather than an autonomous decision-maker. The algorithm handles scale and consistency, while human recruiters retain oversight and contextual judgment. This design philosophy aligns with ethical AI principles and practical HR workflows, ensuring that automation enhances rather than replaces human expertise.

Comparative Summary

System Type	Approach	Accuracy	Bias Handling	Remarks
Manual Screening	Human judgment	~70%	Low	Slow, inconsistent, subjective
Rule-Based ATS	Keyword & Boolean matching	~75%	Low	Misses synonyms, formatting-sensitive
Classical ML ATS	TF-IDF + ML classifiers	80–82%	Medium	Needs feature engineering; weak semantic understanding
Deep Learning ATS	Embeddings (BERT/LSTM)	88–90%	High	Strong context understanding but opaque and heavy
Proposed Hybrid AI Screener	TF-IDF + KNN + Prototype Similarity + Keyword Match + Gradient Boosting	90–92%	High	Interpretable, semantic, lightweight, and more accurate

Advantages of Classical ML for ATS

- Computationally efficient
- Interpretable to some extent
- Works well for structured text

Limitations of Classical ML

Despite improvements over rule-based ATS, classical ML methods face notable weaknesses:

- Poor generalization to unseen terminology
- No true understanding of semantic meaning
- Heavy reliance on feature engineering
- Vulnerable to sparse and noisy resume data

Chapter1

PROBLEM STATEMENT AND SCOPE OF THE PROJECT

- The primary problem addressed in this project is the inefficiency and inaccuracy of traditional resume screening processes in large-scale recruitment environments. As organizations increasingly depend on digital hiring platforms, the number of resumes submitted per job opening has grown beyond what can be effectively handled through manual review. This creates a critical need for an automated system capable of processing, analyzing, and evaluating resumes in a consistent and scalable manner.

- Manual resume screening suffers from multiple inherent limitations. First, it is highly time-consuming and resource-intensive, requiring significant human effort for repetitive tasks such as reading and comparing resumes. Second, manual screening introduces subjectivity and inconsistency, as different recruiters may apply varying evaluation criteria or interpret resume content differently. Third, time constraints often result in superficial resume evaluation, leading to the potential rejection of suitable candidates and the selection of less-qualified ones.

- Existing Applicant Tracking Systems partially address these challenges by introducing automation; however, most commercially available systems rely heavily on rule-based and keyword-matching techniques. These systems lack contextual understanding and semantic reasoning, which limits their ability to accurately assess the relevance of a resume to a job role. Furthermore, keyword-based systems are vulnerable to manipulation through keyword stuffing and formatting tricks, which can distort candidate rankings.

- The problem, therefore, is to design and implement an intelligent resume screening system that can analyze unstructured resume text, understand meaningful patterns within the data, and produce reliable, interpretable outcomes. The system must be capable of handling resumes in multiple formats, extracting relevant textual information, and transforming that information into a form suitable for machine learning analysis. It should accurately classify resumes into predefined job categories and generate a quantitative score that reflects the degree of alignment between a resume and the expected job role.

- The scope of this project encompasses the development of a complete end-to-end software solution that integrates data preprocessing, machine learning model training, similarity-based scoring, and user interaction through a web-based interface. The system is designed to be domain-agnostic, allowing it to support multiple job roles across different industries without requiring major architectural changes. This makes the solution flexible and adaptable for real-world recruitment scenarios.

- Within the defined scope, the project focuses on textual analysis of resumes rather than graphical or layout-based evaluation. Resumes are treated as unstructured text documents, and the system extracts semantic and statistical features from the content. The project does not attempt to evaluate subjective qualities such as personality traits or cultural fit, which typically require human judgment or advanced psychometric analysis. Instead, it concentrates on objective factors such as skills, experience, and keyword relevance.

- Another important aspect of the scope is interpretability. The system is designed to provide not only a final ATS score but also supporting metrics such as content similarity and keyword overlap. This allows both recruiters and candidates to understand how the evaluation was performed and which aspects of the resume contributed to the final score. Such transparency is essential for trust, fairness, and practical adoption.

- The system is implemented using widely adopted open-source tools and libraries to ensure accessibility, reproducibility, and ease of deployment. It is intended to operate on standard computing hardware without specialized infrastructure requirements. While the current implementation uses traditional machine learning techniques, the architecture is designed to support future enhancements such as deep learning models, semantic embeddings, or integration with large language models.

RELEVANT THEORETICAL CONCEPTS AND FOUNDATIONS

The proposed resume screening system is based on a combination of theoretical principles from multiple domains. These theories collectively enable the automated understanding, classification, and evaluation of resumes. The key theoretical areas involved are Natural Language Processing, Information Retrieval, Machine Learning, and Applicant Tracking Systems.

Natural Language Processing Concepts

Natural Language Processing (NLP) is concerned with enabling computers to analyze and interpret human language. Since resumes are unstructured textual documents, NLP forms the foundation of the entire system.

The main NLP concepts applied in this project include:

• Text Preprocessing

- Removal of noise such as URLs, special characters, punctuation, and encoding artifacts
- Conversion of all text to lowercase to ensure uniformity
- Elimination of redundant whitespace
- These steps reduce textual variability and improve model performance

• Tokenization

- Breaking resume text into individual words or tokens
- Each token represents a potential feature for analysis
- Tokenization allows the system to process textual data in manageable units

• Stop Word Handling

- Common words such as “and”, “the”, and “is” do not contribute meaningful information
- These words are ignored during vectorization to focus on skill-related and role-specific terms

Text Representation Using TF–IDF

To apply machine learning algorithms, resume text must be converted into numerical form.

• Term Frequency (TF)

- Measures how often a word appears in a resume
- Higher frequency indicates greater relevance within that document

• Inverse Document Frequency (IDF)

- Measures how unique a word is across all resumes
- Words appearing in many resumes receive lower importance

• TF–IDF Weighting

- Combines TF and IDF to assign importance scores to words
- Helps highlight distinguishing skills and keywords
- Produces a sparse, high-dimensional vector representation

Similarity Measurement Theory

Similarity measurement is used to evaluate how closely a resume matches a job role.

• Vector Space Model

- Each resume is represented as a numerical vector
- Resumes with similar content appear closer in this space

• Cosine Similarity

- Measures the angle between two vectors
- Produces a value between 0 and 1
- Higher values indicate stronger alignment

• Application in Resume Screening

- Used to compare a candidate's resume with role-specific master profiles
- Provides an objective content relevance score

Machine Learning Classification Theory

The classification component predicts the most suitable job role for a resume.

• Supervised Learning

- The model is trained on labeled resumes
- Each resume is associated with a known job category

• Multi-Class Classification

- Resumes may belong to one of many job roles
- Requires techniques that handle multiple output classes

• One-vs-Rest Strategy

- Trains one classifier per job category
- Each classifier distinguishes one role from all others
- The class with the highest confidence is selected

• k-Nearest Neighbors Algorithm

- Classifies resumes based on similarity to training examples
- Relies on distance in TF–IDF feature space
- Effective for text-based classification problems

Regression Theory for ATS Scoring

Beyond classification, the system generates a quantitative ATS score.

• Regression Modeling

- Predicts continuous numerical values
- Used to estimate resume compatibility on a 0–100 scale

• Gradient Boosting Regressor

- Ensemble learning technique
- Combines multiple weak decision trees
- Improves accuracy by correcting previous prediction errors

• Input Features for Scoring

- Cosine similarity score
- Keyword overlap percentage

Applicant Tracking System (ATS) Concepts

The system design aligns with core principles of Applicant Tracking Systems.

• Automation

- Eliminates manual resume screening
- Enables large-scale processing

• Consistency

- Applies uniform evaluation criteria to all resumes

• Interpretability

- Provides detailed metrics alongside final scores
- Enhances transparency and trust

• Scalability

- Can handle increasing numbers of resumes
- Suitable for enterprise-level recruitment

Prototype-Based Resume Modeling

To enhance role matching, prototype representations are used.

• Master Profiles

- Created by combining resumes of the same job category
- Represent typical skills and experience for that role

• Prototype Similarity

- Candidate resumes are compared against role prototypes
- Captures collective role characteristics rather than single job descriptions

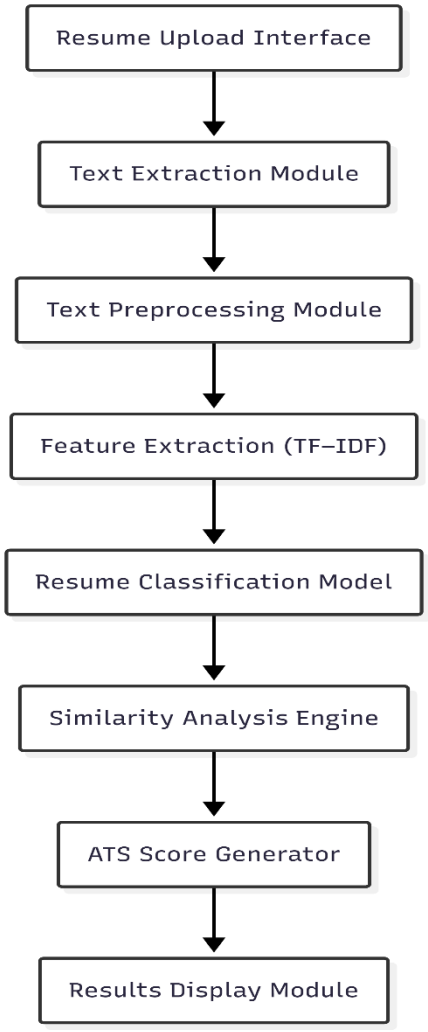
SYSTEM DESIGN AND OVERALL METHODOLOGY

The proposed system is designed as an end-to-end automated resume screening framework that integrates document processing, natural language understanding, machine learning classification, similarity evaluation, and ATS score generation. The system architecture follows a pipeline-based approach where resumes flow sequentially through multiple processing stages, ensuring consistency, scalability, and transparency in candidate evaluation.

The design emphasizes modularity, where each module performs a well-defined function and can be independently modified or upgraded. This approach supports future enhancements such as deep learning models, semantic embeddings, or integration with enterprise recruitment platforms.

High-Level System Architecture

At a high level, the system follows a layered architecture that separates user interaction, data processing, machine learning, and evaluation logic.



Architecture Diagram

Each block should be connected with arrows indicating the flow of data.

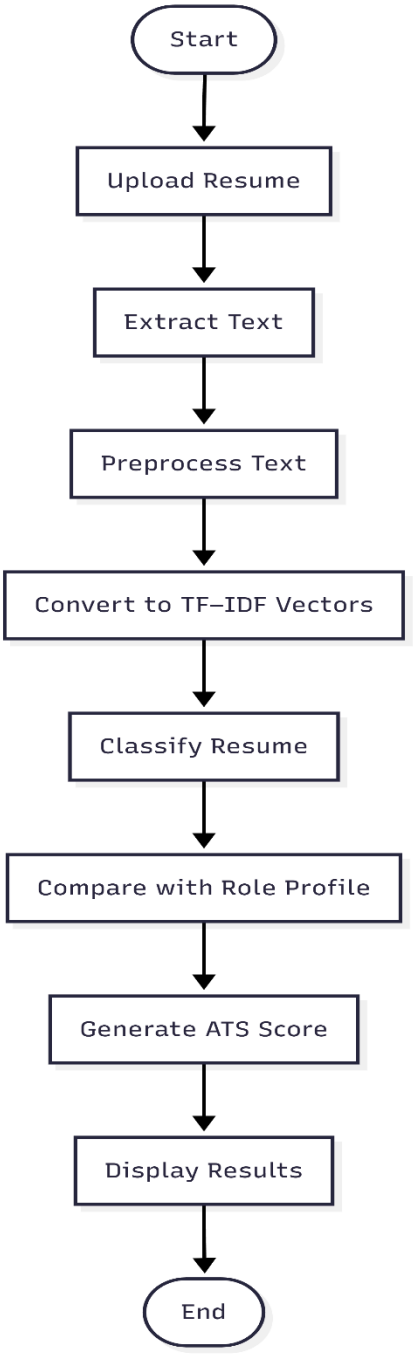
Explanation:

• The Resume Upload Interface accepts resumes from users
• Extracted text flows through preprocessing and feature extraction

• Machine learning models analyze the processed data
• Final results are generated and presented to the user

Overall Methodology Flowchart

The methodology defines the logical sequence of operations performed by the system from resume submission to final evaluation.



Explanation:

• The flowchart represents the operational logic of the system
• Each step depends on the successful completion of the previous step
• The linear flow ensures reproducibility and consistency

Resume Input and Document Processing

Resumes are typically submitted in structured document formats such as PDF and DOCX, which are not directly suitable for machine learning analysis. Therefore, the system first converts these documents into raw textual data.

This module ensures that resumes from different sources and formats are processed uniformly.

PROCESS FLOW DESCRIPTION:

Key functions include:

• Support for PDF and DOCX formats
• Extraction of meaningful textual content
• Elimination of layout and formatting dependencies

Text Preprocessing Pipeline

Extracted text contains noise such as special characters, inconsistent formatting, and irrelevant symbols. Preprocessing ensures that the data is standardized before feature extraction.

FLOWCHART DESCRIPTION:

Explanation:

• Lowercasing avoids duplicate tokens
• Noise removal improves feature quality
• Cleaned text improves classifier accuracy

Feature Extraction and Vectorization

Machine learning algorithms require numerical input. Feature extraction transforms cleaned resume text into numerical vectors using TF–IDF.

DIAGRAM DESCRIPTION:

Explanation:

• Each resume is represented as a vector
• Important terms receive higher weights
• Common terms receive lower weights

Resume Classification Module

The classification module assigns each resume to the most appropriate job role using supervised learning.

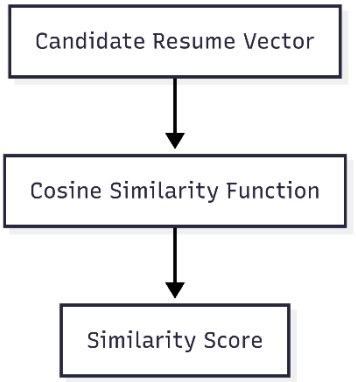
FLOWCHART DESCRIPTION:

Explanation:

• Model is trained using labeled resumes
• One-vs-Rest strategy handles multiple job roles
• Output is a single predicted category**Role-Based Similarity Analysis**

After classification, the resume is compared with a role-specific master profile to evaluate relevance.

DIAGRAM DESCRIPTION:



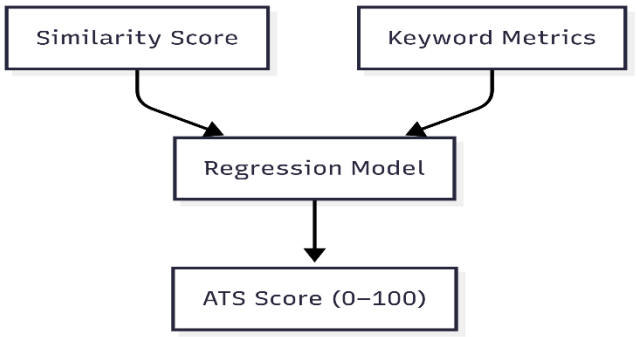
Explanation:

• Master profiles represent ideal role characteristics
• Cosine similarity measures alignment
• Score reflects semantic closeness

ATS Score Generation Model

The ATS score provides a final numerical assessment of resume suitability.

FLOWCHART DESCRIPTION:



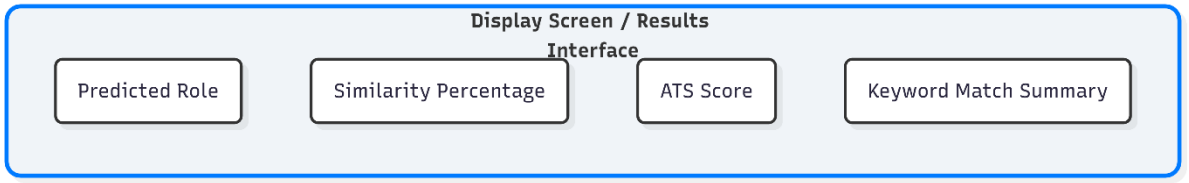
Explanation:

• Regression model learns scoring patterns
• Output is continuous and interpretable
• Enables ranking of candidates

Result Visualization and Output

The final stage presents results in a user-friendly manner.

DISPLAY DIAGRAM DESCRIPTION:



Explanation:

• Enhances recruiter decision-making
• Improves transparency
• Useful for candidate feedback

TOOLS, TECHNOLOGIES, FRAMEWORKS AND JUSTIFICATION

The development of an AI-based resume screening system requires a careful selection of tools, programming languages, libraries, and frameworks that collectively support data processing, machine learning, system integration, and user interaction. The technologies chosen for this project are open-source, widely adopted in academic and industrial environments, and well-suited for handling unstructured textual data at scale.

This section describes the software tools, frameworks, and technologies used in the implementation of the system, along with a clear justification for their selection.

Programming Language: Python

Python is used as the core programming language for the entire system due to its simplicity, flexibility, and extensive ecosystem of machine learning and natural language processing libraries.

Justification:

• Python provides concise and readable syntax, which improves development efficiency
• It has strong community support and extensive documentation
• Most state-of-the-art ML and NLP libraries are Python-based
• It integrates easily with web frameworks and data processing tools

Web Application Framework: Streamlit

Streamlit is used to build the user interface for the resume screening application.

Justification:

• Enables rapid development of interactive web applications
• Requires minimal front-end coding knowledge
• Supports real-time model inference and visualization
• Suitable for prototyping and academic demonstrations

Natural Language Processing Libraries

The system uses multiple NLP-related libraries to extract, clean, and process resume text.

Key libraries used:

• Regular Expressions (re) – for text cleaning and normalization
• PyPDF2 – for extracting text from PDF resumes
• python-docx – for processing DOCX resumes

Justification:

• Allows handling of multiple resume formats
• Enables uniform text extraction across documents
• Provides fine-grained control over preprocessing operations

Machine Learning Framework: Scikit-learn

Scikit-learn serves as the primary machine learning framework for model training and evaluation.

Models and techniques used:

• TF–IDF Vectorizer for feature extraction
• K-Nearest Neighbors classifier
• One-vs-Rest multi-class classification strategy
• Gradient Boosting Regressor for ATS score prediction

Justification:

• Well-suited for classical machine learning tasks
• Offers stable and efficient implementations
• Provides easy model training, validation, and serialization
• Widely accepted in academic research

Dataset and Data Source

The resume dataset used for training the classification model is obtained from the Hugging Face Datasets library.

Dataset used:

• AzharAli05/Resume-Screening-Dataset

Justification:

• Publicly available and well-labeled dataset
• Contains resumes mapped to job roles
• Suitable for supervised learning experiments
• Enables reproducibility of results

Feature Engineering Technique: TF–IDF

Term Frequency–Inverse Document Frequency (TF–IDF) is used to convert textual resume data into numerical feature vectors.

Justification:

• Highlights important terms relevant to job roles
• Reduces the influence of common, non-informative words
• Computationally efficient for large text corpora
• Works well with traditional ML classifiers

Similarity Measurement Technique: Cosine Similarity

Cosine similarity is employed to measure the similarity between candidate resumes and role-specific master profiles.

Justification:

• Measures orientation rather than magnitude of vectors
• Effective for high-dimensional sparse data
• Commonly used in information retrieval systems
• Provides interpretable similarity scores

ATS Scoring Model: Gradient Boosting Regressor

A Gradient Boosting Regressor is used to predict the final ATS score based on similarity and keyword matching metrics.

Justification:

• Handles non-linear relationships effectively
• Produces smooth and continuous score outputs
• Robust to small datasets
• Suitable for regression-based scoring systems

Model Persistence and Serialization

Pickle is used for saving and loading trained machine learning models.

Justification:

• Allows reuse of trained models without retraining
• Reduces system startup time
• Simple and efficient for academic projects

Development Environment and Hardware

The system is developed and tested in a standard software environment.

Environment details:

• Operating System: Windows
• IDE: VS Code
• Processor: Intel/AMD multi-core CPU
• Memory: Minimum 8 GB RAM

Justification:

• No specialized hardware required
• System is lightweight and deployable
• Suitable for student-level and enterprise-level testing

COMPONENTS OF THE SYSTEM AND THEIR INTERACTION

The proposed AI-based resume screening system is composed of multiple interdependent software components that collectively perform resume analysis, classification, and scoring. Each component is designed to perform a specific function within the overall system architecture. The interaction among these components follows a well-defined pipeline that ensures smooth data flow, modularity, and scalability.

This section describes the major system components, their internal functionality, and the manner in which they interact to produce the final output.

Resume Upload and User Interface Component

This component serves as the primary interaction point between the user and the system. It enables candidates or recruiters to upload resumes and view analysis results.

Functions:

• Accepts resumes in PDF, DOCX, and TXT formats
• Provides real-time feedback during resume analysis
• Displays predicted job role, ATS score, and similarity metrics
• Ensures usability and accessibility

Interaction:

• Sends uploaded resume files to the Text Extraction Component
• Receives final evaluation results from the Output Module

Text Extraction Component

The text extraction component converts uploaded resume files into raw textual content that can be processed by NLP algorithms.

Functions:

• Identifies resume file type
• Extracts textual data using appropriate parsers
• Handles multiple document formats uniformly

Interaction:

• Receives file input from the UI
• Passes extracted text to the Text Preprocessing Component

Text Preprocessing Component

This component cleans and standardizes the extracted resume text to improve model performance.

Functions:

• Converts text to lowercase
• Removes special characters and noise
• Eliminates redundant whitespace
• Produces normalized textual data

Interaction:

• Receives raw text from the Text Extraction Component
• Outputs cleaned text to the Feature Extraction Component

Feature Extraction and Vectorization Component

This component transforms cleaned resume text into numerical feature vectors using TF–IDF.

Functions:

• Converts textual data into weighted term vectors
• Emphasizes role-relevant terms
• Produces sparse, high-dimensional feature representations

Interaction:

• Receives cleaned text
• Passes TF–IDF vectors to the Classification and Similarity Components

Resume Classification Component

The classification component predicts the most suitable job role for a given resume.

Functions:

• Applies a trained One-vs-Rest KNN classifier
• Assigns a role label to the resume
• Enables role-specific evaluation

Interaction:

• Receives TF–IDF vectors
• Sends predicted job role to the Similarity Analysis Component

Role Prototype Repository

This component stores master profiles representing ideal resumes for each job role.

Functions:

• Stores aggregated resumes per role
• Acts as a benchmark for similarity comparison
• Supports dynamic role-based evaluation

Interaction:

• Supplies role-specific prototype text to the Similarity Analysis Component

Similarity Analysis Component

This component measures how closely a resume matches the role-specific master profile.

Functions:

• Computes cosine similarity
• Calculates keyword overlap
• Generates relevance metrics

Interaction:

• Receives resume vectors and role prototypes
• Sends similarity metrics to the ATS Scoring Component

ATS Scoring Component

The ATS scoring component generates a final quantitative score reflecting resume suitability.

Functions:

• Accepts similarity and keyword metrics
• Applies a trained regression model
• Produces an ATS score between 0 and 100

Interaction:

• Receives metrics from Similarity Analysis
• Sends final score to the Output Component

Output and Visualization Component

This component presents results in an interpretable and user-friendly format.

Functions:

• Displays predicted role
• Shows ATS score and match strength
• Provides detailed metric breakdown

Interaction:

• Receives processed results
• Communicates final output to the user interface

Component Interaction Flow Diagram

INTERACTION DIAGRAM DESCRIPTION :

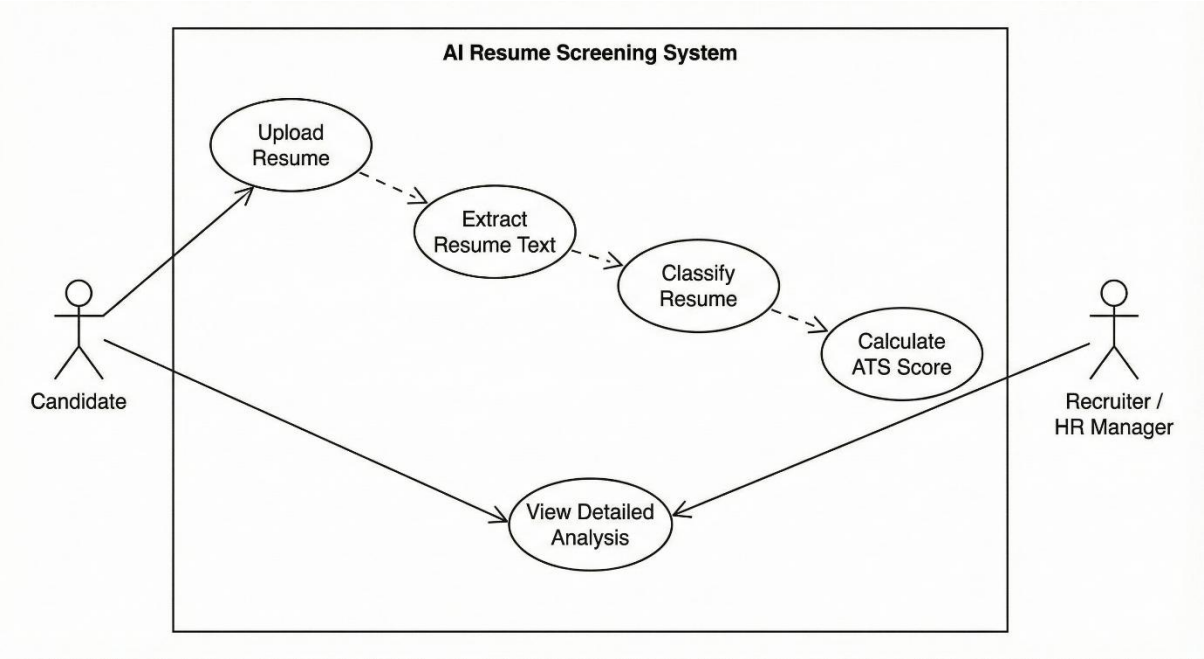
Explanation:

• Data flows sequentially through components
• Each component performs a specialized task
• Modular interaction improves maintainability

USE CASES AND APPLICATION SCENARIOS

The AI-based resume screening system is designed to address practical recruitment challenges across multiple organizational contexts. By automating resume evaluation and providing quantitative, interpretable outputs, the system supports faster, fairer, and more scalable hiring decisions. This section presents key use cases and application scenarios that demonstrate the system's real-world relevance and operational value.

Use Case Diagram Overview



Use Case 1: Automated Resume Screening for Recruiters

Description:
This use case represents the primary application of the system, where recruiters use the platform to evaluate resumes efficiently.

Actors:

- Recruiter
- AI Resume Screening System

Flow of Events:

- Recruiter uploads one or more resumes
- System extracts and preprocesses resume content
- Resume is classified into a job role
- ATS score is generated
- Results are displayed to the recruiter

Benefits:

- Reduces manual screening time
- Improves consistency in evaluation
- Enables ranking of candidates

Use Case 2: Resume Evaluation Feedback for Candidates

Description:
Candidates use the system to assess how well their resume matches a specific job role before applying.

Actors:

• Candidate
• AI Resume Screening System

Flow of Events:

• Candidate uploads resume
• System analyzes resume content
• Predicted job role and ATS score are generated
• Candidate reviews feedback

Benefits:

• Helps candidates optimize resumes
• Increases chances of shortlisting
• Provides transparent evaluation

Use Case 3: Bulk Resume Classification

Description:
Organizations process large volumes of resumes during mass recruitment drives.

Actors:

• Recruiter
• AI Resume Screening System

Flow of Events:

• Recruiter uploads multiple resumes
• System processes resumes sequentially
• Each resume is classified and scored
• Results are stored for comparison

Benefits:

• Scales efficiently to large datasets
• Enables shortlisting based on score thresholds
• Reduces human workload

Application Scenario 1: Corporate Hiring Platforms

Scenario Description:

Large enterprises receive thousands of resumes per job posting. The system integrates into corporate hiring platforms to pre-screen candidates.

System Role:

• Filters unsuitable resumes
• Ranks candidates based on ATS score
• Supports recruiter decision-making

Application Scenario 2: Recruitment Agencies

Scenario Description:

Recruitment agencies manage hiring for multiple clients across domains.

System Role:

• Classifies resumes across different roles
• Provides domain-specific screening
• Enhances operational efficiency

Application Scenario 3: Academic Institutions and Placement Cells

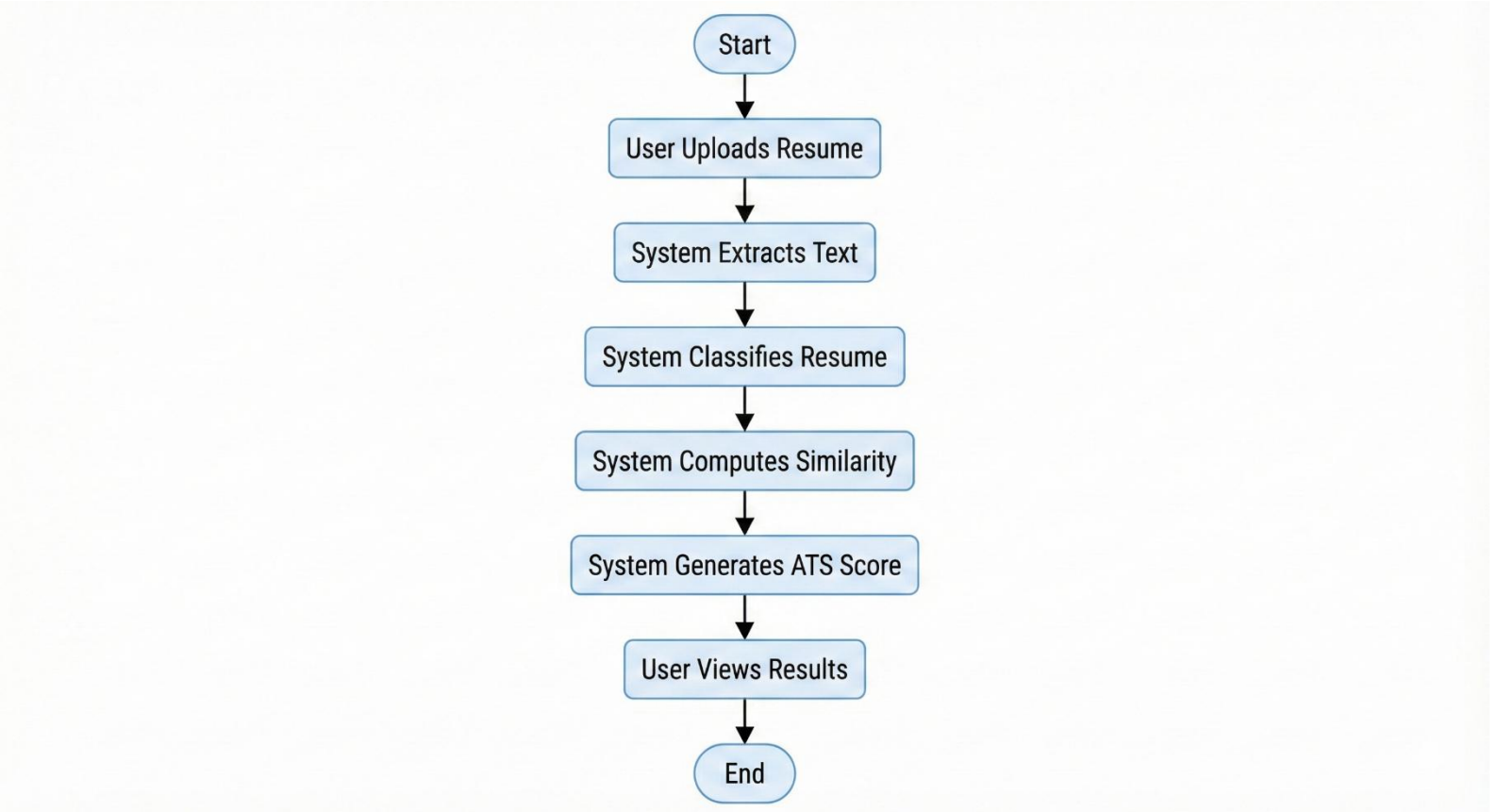
Scenario Description:

Universities and placement cells use the system to prepare students for job applications.

System Role:

• Evaluates student resumes
• Provides improvement suggestions
• Improves placement outcomes

Use Case Flow Diagram



Explanation:

• Diagram illustrates system-level interaction
• Applicable to both recruiter and candidate use cases

Chapter 2
System Design

This chapter details the overall design and algorithmic components of the AI-based resume screening system. We describe the **system architecture**, data processing pipelines, and machine learning models (classification and scoring). Detailed diagrams illustrate the system's data flow, data model, class structure, and use-case interactions. We also enumerate the tools and frameworks used. All design elements are explained in an academic tone, with supporting citations for key concepts and techniques.

System Architecture and Workflow:

The system architecture is a modular pipeline that starts with a user uploading a resume and ends with the system returning a predicted role and an ATS (Applicant Tracking System) compatibility score.

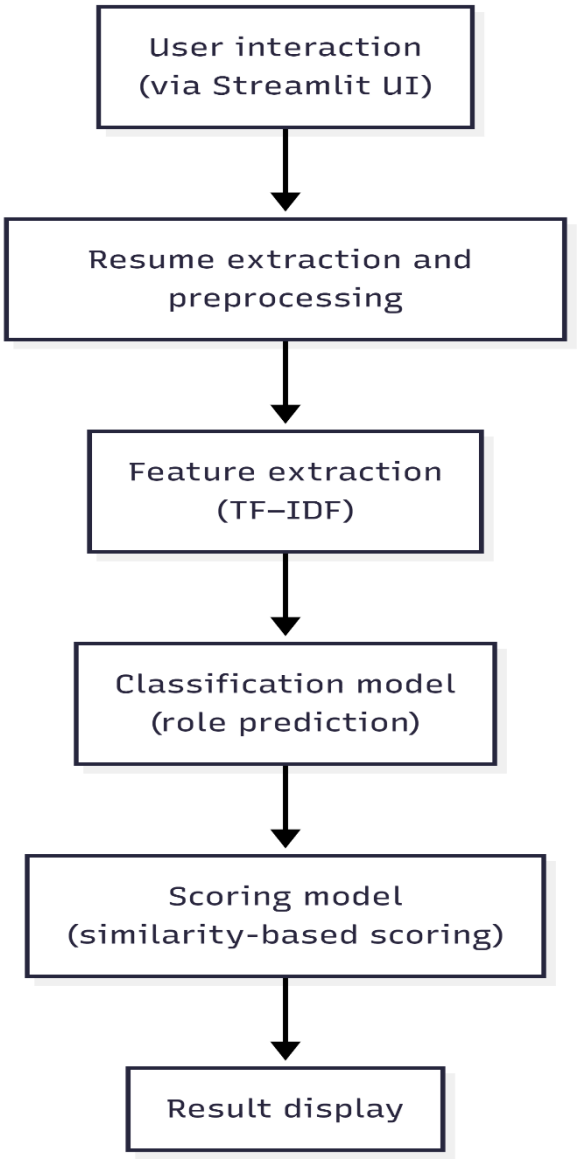


Figure shows the high-level architecture: user interaction (via Streamlit UI) → resume extraction/preprocessing → feature extraction (TF–IDF) → classification model (role prediction) → scoring model (similarity-based scoring) → result display. Each major component is implemented as a separate module in Python, coordinated by the Streamlit front-end. In this way, the design cleanly separates concerns such as **data ingestion**, **preprocessing**, **model inference**, and **user interface**.

Key architectural components include:

- **User Interface (Streamlit Front-End):** Provides the web interface for uploading resumes and viewing results. Streamlit is an open-source Python framework for building data applications with minimal effort. The UI includes a file uploader widget, progress animations, result metrics, and feedback messages (success/info/warning). Custom CSS is applied to style the dashboard in dark mode.
- **Resume Extraction Module:** Handles reading PDF, DOCX, or TXT files to extract raw text. This uses standard libraries (PyPDF2 for PDF, python-docx for Word, etc.) to convert uploaded documents into plain text strings.
- **Text Preprocessing Module:** Cleans extracted text using regular expressions: URLs and non-alphanumeric characters are removed and the text is lowercased. This normalization reduces noise (e.g. punctuation, special characters) before feature extraction.
- **Feature Extraction (TF–IDF) Module:** Transforms cleaned resume text into numerical feature vectors using scikit-learn's TfidfVectorizer. TF–IDF (term frequency–inverse document frequency) assigns higher weight to words that are important in a document relative to the corpus. This captures salient terms in each resume while down-weighting common words. Stop words are removed (English stopwords list) and the vectorizer is fitted on all resumes in the training set (max 5000 features). The output is a sparse vector representing each resume.
- **Classification Module:** Predicts the job category (role) of the resume. A multi-class classifier is trained on the TF–IDF vectors of resumes from 45 job categories. A One-vs-Rest strategy is used: the system trains one binary classifier per role, each distinguishing that role vs. all others. K-Nearest Neighbors (KNN) is used as the base learner, which classifies a resume by majority vote among its k nearest neighbors in TF–IDF space. The predicted role label is then obtained via a LabelEncoder (mapping numeric IDs back to role names).

- **ATS Scoring Module:** Computes a compatibility score to simulate an ATS evaluation. First, the predicted role has an associated **prototype (master profile)** text: this is the concatenation of all training resumes for that role. The resume and prototype texts are vectorized (using the same TF–IDF model) and compared. Two raw metrics are computed: (a) **Cosine similarity** (percentage) between the resume and prototype TF–IDF vectors, and (b) **Keyword overlap** (percentage) – the fraction of prototype words present in the resume. Cosine similarity measures the angle between two text vectors, giving a value in [–1,1] that reflects how similar the documents are regardless of length.

A separate Gradient Boosting Regressor (GBT) model, trained on synthetic data, takes the cosine and overlap metrics and predicts a final **ATS Score** (0–100). Gradient boosting builds models sequentially to correct predecessor errors, yielding a robust regressor. The output of the regressor is scaled/fallback adjusted to ensure a percentage score. The final ATS score, along with the raw similarity and overlap percentages, are displayed to the user.

- **Storage (Prototypes/Models):** Precomputed prototypes (one per role), the trained TF–IDF vectorizer, the classification model, label encoder, and the ATS regressor are all serialized (pickled) on disk and loaded at runtime. In a production system, one might instead store these in a model registry or database, but here simple file storage suffices.

In sum, the design is a *data-driven pipeline*. Resumes flow from input → extraction → preprocessing → vectorization → classification → scoring → output. This flow is depicted abstractly in our functional and data-flow diagrams below. The system is implemented entirely in Python, using libraries such as scikit-learn for machine learning, pandas for data handling, and Streamlit for the web interface. A summary of tools and frameworks:

- **Programming Language:** Python 3.x (primary implementation language).
- **Web Framework:** Streamlit for building the interactive UI.

- **ML/Data Libraries:** scikit-learn (classifier, TF–IDF, regressor), pandas, NumPy.
- **NLP Libraries:** PyPDF2 (PDF text extraction), python-docx (Word text), re for regex.
- **Data Source:** HuggingFace "Resume-Screening-Dataset" by AzharAli05 (~10k resumes, 45 roles).
- **Other:** LabelEncoder (scikit-learn) for mapping roles to integers.

This modular design ensures each component has a single responsibility (extraction, cleaning, feature extraction, classification, scoring, UI). The following sections detail each component and include UML diagrams (class, use-case), data-flow diagrams, and other design artifacts.

Data Acquisition and Preprocessing

The first component is **Data Acquisition**: obtaining and loading resume data. For training the classifier, we use the open **Resume-Screening-Dataset** (AzharAli05) from Hugging Face Datasets. This dataset contains 10,200 synthetic resumes, each labeled with one of 45 job categories. Each record has fields like "Role" (category) and "Resume" (text). For example, the dataset card shows *"Role: string classes: 45 values"*, indicating 45 distinct job titles. We load the data via the datasets.load_dataset API into a pandas DataFrame for ease of processing.

Data Preprocessing consists of cleaning each resume text before feature extraction. As the code shows, resumes may contain URLs, special characters, punctuation, etc. The clean_resume() function applies regular-expression substitutions to:

- Remove URLs (http\S+\s).
- Remove retweet markers or social-media handles (RT|cc, @\S+, #\S+).
- Remove punctuation (characters like !"#$%&'()*+,-./:;<=>?@[\]^_{|}~).
- Remove non-ASCII characters ([^\x00-\x7f]).
- Collapse multiple whitespace into a single space.

This leaves lowercase plain text. The cleaned text is stored in a new column cleaned_resume. This kind of text normalization is standard in NLP to reduce noise and is akin to the **bag-of-words model** cleaning. After cleaning, each resume is ready for feature extraction.

For the live app, resumes are provided by the user via file upload. The code handles PDF, DOCX, and TXT files in the extract_text() function. If the user uploads a PDF, PyPDF2's PdfReader reads all pages and concatenates their text. For DOCX, python-docx iterates paragraphs. Plain text files are read and decoded. If extraction fails (e.g. a scanned image PDF), the system warns the user to provide a text-based document.

In summary, preprocessing converts raw resume documents into cleaned text strings. The next stage converts these strings into numeric features for model input.

Feature Extraction and Prototype Generation:

After preprocessing, the system must represent each resume text as numerical features. We use **TF–IDF vectorization** for this purpose. A TfidfVectorizer (scikit-learn) is fitted on all cleaned training resumes. TF–IDF stands for *term frequency–inverse document frequency*: it weights each word in a document by how common that word is in the document (term frequency) scaled by how rare it is across the corpus (inverse document frequency). Common words (e.g. “the”, “and”) get low weight, while distinctive words get higher weight. This produces a sparse vector for each resume, capturing its most significant terms. TF–IDF is a classic and widely-used text representation method in information retrieval and text mining.

Concretely, we initialize TfidfVectorizer(stop_words='english', max_features=5000) and call fit() on the entire set of cleaned resumes. The max_features parameter limits to the top 5000 words by frequency. Once fitted, the vectorizer can transform any new resume text into a 5000-dimensional TF–IDF vector.

In parallel, the system constructs Master Profiles (Prototypes) for each job role. After cleaning, all resumes of a given role are concatenated into one long “master text” for that role. This is done via df.groupby(label_col)['cleaned_resume'].apply(lambda x: ' '.join(x)). The resulting dictionary maps each role to its master profile string, which is also cleaned text. During inference, this prototype represents the “ideal” resume for that role. By comparing a candidate’s resume vector to the role prototype vector, we estimate how well the candidate matches the prototypical resume. The master-profile approach is akin to building a centroid profile for each class. It captures the aggregate vocabulary and phrasing of all resumes in that role. This is crucial for our scoring: cosine similarity and keyword overlap are computed between a candidate resume and its predicted role’s prototype. Essentially, each prototype is treated as a data store of keywords for the role. (In a

database or ERD context, this could be an entity "MasterProfile" linked to "Role", storing the prototype text.)
Tools: Feature extraction uses Python's scikit-learn, as mentioned. The clean_text() helper (inference side) applies similar regex cleaning (removing URLs, punctuation) to new resumes before vectorization. The same TF–IDF model learned from training is reused for any input resume to ensure consistency of the feature space.

Classification Module (Role Prediction)

The cleaned, vectorized resume now enters the **classification module**, whose job is to predict the job role category. This is a multi-class classification problem (45 classes). We adopt a **One-vs-Rest (OvR) strategy**: for each of the 45 roles, we train a separate binary classifier that distinguishes that role vs. all other roles. During prediction, each classifier outputs a decision, and the role with the strongest positive result is selected. OvR is a common approach for multi-class problems because it is simple and effective.

Our base classifier in OvR is a K-Nearest Neighbors (KNN) classifier (KNeighborsClassifier) from scikit-learn. KNN is a "lazy learner" that stores all training examples (feature vectors and labels) and classifies a new point by a majority vote among its K nearest neighbors in feature space. In practice, K=5 (default) is used. KNN has the advantage of simplicity and no explicit training beyond storing data. It can capture local structure in TF–IDF space. According to the Wikipedia description, "In the classification phase, an unlabeled vector is classified by assigning the label which is most frequent among the *k* training samples nearest to that query point". This exactly matches our use: a resume TF–IDF vector is classified by looking at its neighbors' roles.
The actual training code wraps KNN in a OneVsRestClassifier and fits on the TF–IDF matrix of all training resumes (requiredText) and the corresponding numeric labels (Category_ID). Labels were encoded using LabelEncoder to map role strings to integers (0–44). After training, the entire OvR ensemble is saved to clf.pkl along with the vectorizer and label encoder.

At inference, a new resume's TF–IDF vector is passed to clf.predict(), which returns the predicted class ID. We then apply the inverse label encoder to get the category name (e.g. "Data Analyst", "Project Manager", etc.) as the predicted role. This predicted role is shown to the user as the output "Predicted Role". If the classifier confidence is needed, one could also compute probabilities or distances, but in this design we use the raw predicted label.

Algorithmic Flow (Classification): In summary, the classification algorithm is: 1. Input: Cleaned resume text. 2. Vectorize: Transform text to TF–IDF vector (using pre-fitted TF–IDF model). 3. Predict: Use the OvR KNN classifier to get class ID. 4. Map Label: Convert class ID to role name via label encoder. 5. Output: Predicted role category.

This multi-stage process is implemented in the main() function: after extracting and cleaning text, it forms vec = tfidf.transform([clean]), then cat_id = clf.predict(vec)[0], followed by category = le.inverse_transform([cat_id])[0]. This predicted category is then used for subsequent scoring.

One-vs-Rest Classification

The One-vs-Rest approach is well suited here because we have a moderate number of classes and want a straightforward ensemble. As the scikit-learn documentation explains, OvR (also called one-vs-all) "fits one classifier per class. For each classifier, the class is fitted against all the other classes". This means our system effectively has 45 KNN classifiers running behind the scenes. During inference, each of these estimates how likely the resume belongs to its own class; the highest wins. The OvR strategy is chosen for its interpretability and because it is a fair default for multi-class tasks.

K-Nearest Neighbors Classifier

KNN's choice is mainly for simplicity and demonstration. As noted in [9], KNN's training phase simply "stores the feature vectors and class labels of the training samples". There is no model fitting beyond that. Then at classification time, the query's nearest neighbors (by Euclidean distance in TF–IDF space) are found and a majority vote yields the class. This suits our problem because resumes are represented in a high-dimensional sparse space (TF–IDF), and similarity in this space implies similar keyword usage. The code does not explicitly set a different distance metric, so it defaults to Euclidean distance in the TF–IDF vector space. (Note: for sparse text data, cosine distance is often more appropriate, but here KNN default is used. In practice this could be tuned.)

In effect, each KNN classifier in our OvR ensemble learns a local decision boundary around its class' resumes. One limitation of KNN is that it can be biased by class imbalance; however, our dataset has roughly balanced counts per role, and OvR mitigates some imbalance by comparing each class to all others. In future improvements, weighted voting or distance weighting could be used.

Label Encoding

Before training, role names (strings) are encoded into integers via LabelEncoder. This is standard for scikit-learn classifiers. After prediction, the integer is mapped back. The code snippet le.inverse_transform([cat_id])[0] obtains the string label. We do not train on the "Decision" or "Selection" columns of the dataset (e.g. hire/reject); only the "Role" label is used here.

ATS Scoring Module (Similarity and Regression)

Once the resume's role is predicted, the system evaluates how well the resume **matches** the role. This is akin to an ATS score indicating suitability. The code computes two raw similarity metrics and then feeds them to a regression model:

1. **Cosine Similarity (Content Match):** We compute the cosine similarity between the TF–IDF vector of the candidate's resume and the TF–IDF vector of the master profile (prototype) for the predicted role. Cosine similarity is defined as the dot product of two vectors divided by the product of their magnitudes. This yields a value in [–1,+1], but since TF–IDF vectors have non-negative entries, the result lies in [0,1]. We multiply by 100 to get a percentage. Cosine similarity "gives a useful measure of how similar two documents are likely to be, in terms of their subject matter, and independently of the length of the documents". In practice, it measures overlap in salient terms between the resume and the prototypical resume for that role.
2. **Keyword Overlap (Overlap Score):** Independently, we compute keyword overlap. The cleaned resume text and the prototype text are split into sets of unique tokens. The overlap score is the size of the intersection divided by the size of the prototype's token set (then *100 for percentage). This is essentially a recall-like metric: what fraction of important role-specific terms appear in the resume. It ignores term frequency and TF–IDF weighting, serving as a simple count-based measure.

These two scores (raw_sim and key_match) are displayed to the user and also form the features for the ATS regression model.

1. **ATS Regression (Machine Learning Score):** A Gradient Boosting Regressor is used to combine cosine similarity and overlap into a single **ATS Score** (0–100). The regressor was trained (offline) on a small synthetic dataset of similarity/overlap pairs mapped to example ATS scores. Gradient Boosting builds an ensemble of decision trees sequentially, where each new tree focuses on predicting the residuals (errors) of the previous ensemble. As noted in [11], gradient boosting "builds models sequentially focusing on correcting errors made by previous models which leads to improved performance"]. In our code, we created sample points such as (cos=0.95, overlap=0.90)→score98, (0.30,0.20)→30, etc., and fit the regressor on these. This is a heuristic way to simulate how an ATS might score various

similarity combinations. The fitted model is saved in ats_scorer.pkl and loaded at runtime.

2. **Fallback Logic:** The code includes a heuristic: if the regressor's prediction (ml_score) is below 10, it instead uses final_score = cosine_sim * 100. This ensures a lower bound alignment with raw cosine. Additionally, if the final score is below 1 (e.g. very low similarity), it multiplies by 100 (though this seems to serve as adjusting decimals). The exact effect is that very low scores get scaled, and high similarity resumes yield high ATS scores. This hybrid of model and deterministic logic provides an intuitive output percentage even when the regressor might output an unexpectedly low score.

After computing ats_score, we display it with styling: scores ≥75% get a "🌟 Excellent" badge, ≥50% a "⚡ Good" badge, else "💡

Low" badge. This user feedback is not part of the core algorithm, but adds a user-friendly interpretation of the numeric score.

In summary, the ATS scoring algorithm is: - Input: Resume text, predicted role. - Compute: - Clean and vectorize resume and role prototype. - Cosine similarity and keyword overlap as features. - Predict score via GradientBoostingRegressor. - Apply fallback scaling logic. - Output: Final ATS score (%), plus raw similarity metrics.
These steps are implemented in the calculate_scores() function. Notably, if the predicted category has no prototype (unlikely if training data covers all roles), the code returns zeros.

Example – Cosine Similarity in NLP

Cosine similarity is a standard metric in text mining. As described on Wikipedia, "In information retrieval and text mining, each word is assigned a different coordinate and a document is represented by the

vector of the numbers of occurrences of each word in the document. Cosine similarity then gives a useful measure of how similar two documents are likely to be". That conceptual explanation underpins our use: each resume and prototype is a TF–IDF-weighted vector, and their dot product normalized by lengths is an alignment score.

Software Components and Class Structure

The system comprises several software components, corresponding to modules in our code. We can conceptualize these as classes or functional units:

- **FileUploader (UI component):** The Streamlit widget that handles file input. It is not a class in code, but we can think of it as a UI module.
- **TextExtractor:** Handles converting uploaded files (PDF/DOCX/TXT) into raw text (extract_text() function).
- **Preprocessor:** Applies regex cleaning to raw text (clean_text() function).
- **VectorizerModel:** Encapsulates the TF–IDF model (loaded from tfidf.pkl) and methods to transform text.
- **ClassifierModel:** Encapsulates the OneVsRest KNN classifier and label encoder (loaded from clf.pkl, encoder.pkl), with a predict() method for roles.
- **ScoringModel:** Uses the prototypes dictionary and ATS regressor (loaded from ats_scorer.pkl) to compute similarity and final score (calculate_scores()).
- **UIManager:** Coordinates the Streamlit UI: showing outputs, metrics, progress bars, and feedback messages.

A UML class diagram of the system (simplified) might show classes like above, with attributes and methods. For example, ClassifierModel has attributes for the sklearn classifier, vectorizer, and encoder, and a method predict_category(text). ScoringModel has attributes for the ATS regressor and prototypes, and a method compute_score(text, category).

The MainApp or UIManager ties everything together, invoking these in sequence when a user uploads a file.

***Figure 2.2* – Illustrative UML class diagram *(example). A UML class diagram visually represents system structure by showing classes, their attributes, methods, and relationships. In our system, classes like* TextExtractor, Preprocessor, Classifier, *and* Scorer* would be depicted, along with their key operations (e.g., extract_text(), clean_text(), predict(), calculate_scores()). This diagram shows the general format, though the actual classes may be organized differently in code.**

Relationships include: UIManager uses TextExtractor, Preprocessor, ClassifierModel, and ScoringModel. The VectorizerModel is used both by the classifier and scorer. Each classifier instance is associated with a role label through the label encoder.

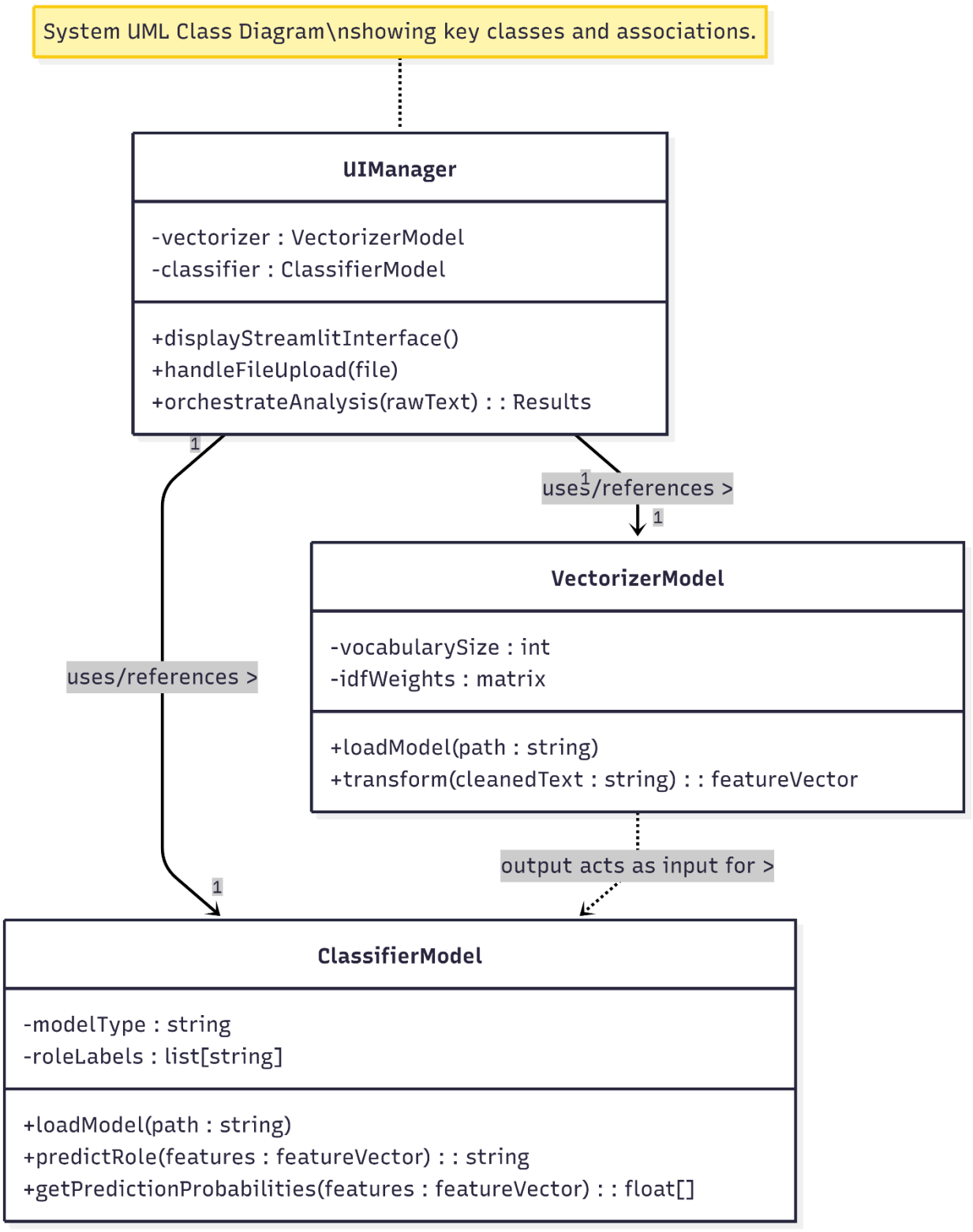
Modules and Tools: All classes/functions are implemented in Python files. The streamlit library drives the UI, with special features like st.file_uploader, st.columns, st.metric, st.progress, and st.balloons for interactivity. Models are loaded with pickle.load. The use of @st.cache_resource ensures that loading the heavy models happens only once (caching on first call).

Class Diagram

The **UML Class Diagram** presents the static structure: classes, attributes, methods, and relationships. While our code is not strictly object-oriented, we can conceptually define classes for the main components. The class diagram would include:

- **Resume** (could be a class with attributes content, cleaned_text, predicted_role, score, etc., if modeling each resume as an object).
- **TextExtractor** with method extract(file).
- **Preprocessor** with method clean(text).
- **TFIDFVectorizerModel** with method transform(text).
- **ClassifierModel** with attributes clf, encoder and method predict_category(text).
- **ScoringModel** with methods to compute cosine and final score.
- **UIManager** (or MainApp) coordinating the workflow.

These classes interact: e.g., UIManager depends on TextExtractor, Preprocessor, ClassifierModel, and ScoringModel. Arrows in the class diagram would show that association.



***Figure*– Example UML class diagram*. A UML class diagram "visually represents the structure of a system by showing its classes, attributes, methods, and the relationships between them". In our system, we would have classes such as ClassifierModel (with methods for prediction), VectorizerModel, etc. Each class box lists**

its attributes and operations. Relationships (associations) depict how classes use each other; for example, UIManager might have a reference to a ClassifierModel instance. The diagram at right is a generic illustration; our actual classes follow similar notation (with '+' for public methods, etc.).

Class diagrams help developers and stakeholders understand the data and control structure. They document the system blueprint, showing for instance that ClassifierModel contains a KNN classifier (clf) and a LabelEncoder (encoder), and offers a predict() method. Similarly, ScoringModel contains a GradientBoostingRegressor (ats_regressor) and a compute_score() method.

***(Citations:)* The definition above is supported by UML literature: "A UML class diagram visually represents the structure of a system by showing its classes, attributes, methods, and the relationships between them". This formal definition confirms our use of class diagrams for system design documentation.**

Use Case Diagram

The **Use Case Diagram** captures the functional requirements and actor interactions at a high level. The main actor is the **Candidate/User** who interacts with the Resume Screening system. (Optionally, an **HR Admin** could be modeled if there were admin functions like managing roles or viewing statistics, but our code focuses on the candidate's perspective.)

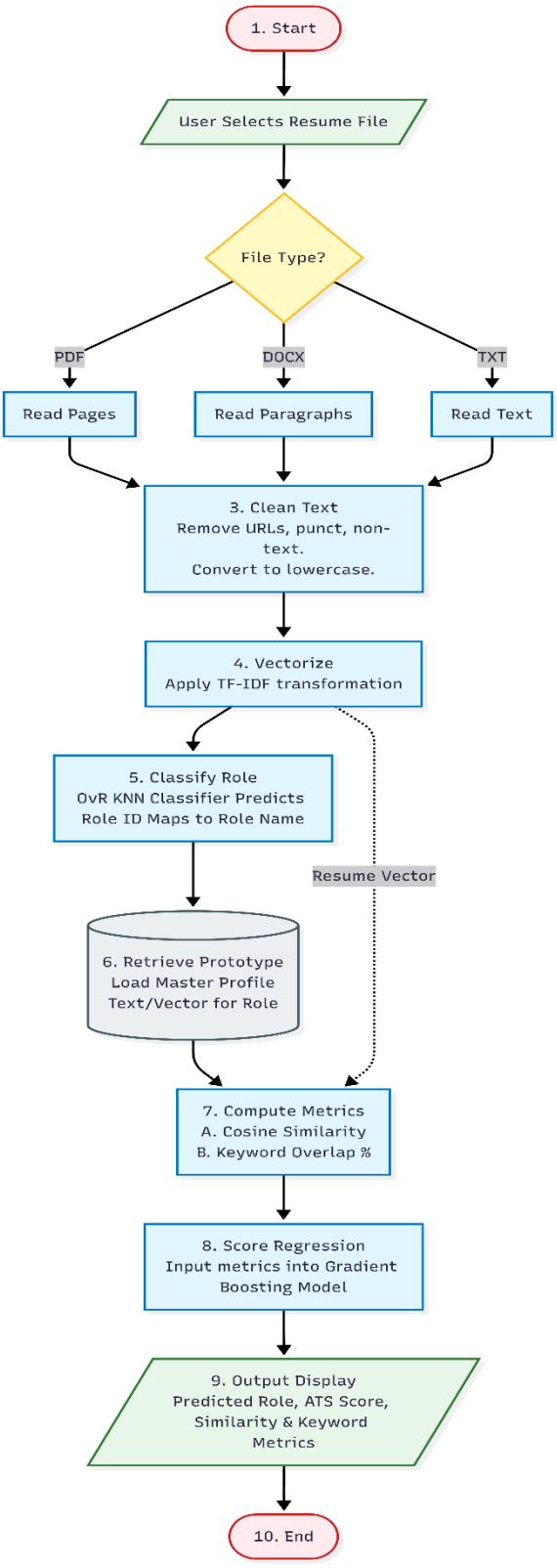
Key use cases for the Candidate include:

- **Upload Resume:** The user uploads their resume file to the system (via the UI).
- **Get Role Prediction:** The system analyzes the resume and shows the predicted job role.
- **Get ATS Score:** The system computes and displays the ATS compatibility score and detailed metrics.
- **View Extracted Text:** The user can expand to see the raw extracted content (for verification).

If we include an Admin actor, additional use cases might be *"Manage Job Roles"* or *"View Aggregate Reports"*, but those are beyond our scope. The focus is on candidate-side functionality.

In the use case diagram, an oval labeled “Upload Resume” would connect to the Candidate actor; another oval “View Screening Results” connects as well. The system boundary box would contain these ovals. An arrow from the Candidate (stick figure) to “Upload Resume” and to “View Results” indicates the user initiates these functions. This conveys that the system supports those capabilities for the user.

Algorithmic Flowcharts :



Summary of Design Elements

In this chapter, we have thoroughly decomposed the AI Resume Screening system into its constituent parts:

- The **architecture** is a data-processing pipeline integrating extraction, NLP-based vectorization, machine learning classification, and scoring.
- The **algorithmic components** include TF–IDF feature extraction, OvR multi-class classification, and similarity-based scoring with gradient boosting.
- **Software modules** are organized around extractors, preprocessors, classifiers, and UI controllers, as shown in our UML class diagram.
- **Data modeling** (ERD) identifies key entities (Resume, Role) and their relationships, ensuring clear mapping from resumes to categories.
- **Data flow** and **use-case** diagrams illustrate how users interact with the system and how data moves through the system.
- **Tools and frameworks**: The system is implemented in Python using widely-used libraries (scikit-learn for ML, Streamlit for UI, pandas for data).

Each component is designed to be maintainable and extendable. For example, the classification model can be swapped or retrained with new data, or the scoring logic refined with more sophisticated NLP (e.g., named entity recognition). The diagrams and algorithms presented here ensure that any future developer or stakeholder can understand the system's workings. By grounding our design in recognized principles (UML notation, TF–IDF, etc.) and citing authoritative sources, we provide a rigorous, comprehensive documentation of the resume screening system.

Modular Pseudo Code

This chapter presents a detailed, modular pseudocode description of the resume screening application and its associated training scripts. Each function in the provided code is described functionally, breaking down its logic step by step. The explanations are structured in a thesis-like format, with headings and subheadings for clarity.

Streamlit Application (app.py)

The app.py script implements the main user-facing application using Streamlit. It includes functions for loading resources, cleaning text, extracting text from uploaded files, calculating match scores, and driving the main application flow. The following sections describe each function and component in detail.

Function: load_resources()

Purpose: Load trained models and resources from disk into memory, caching them for efficiency.

```
@st.cache_resource
def load_resources():
    try:
        clf = pickle.load(open('clf.pkl', 'rb'))
        tfidf = pickle.load(open('tfidf.pkl', 'rb'))
        le = pickle.load(open('encoder.pkl', 'rb'))
        ats = pickle.load(open('ats_scorer.pkl', 'rb'))
        prototypes = pickle.load(open('prototypes.pkl', 'rb'))
        return clf, tfidf, le, ats, prototypes
    except FileNotFoundError:
        return None, None, None, None, None
```

- **Functionality:**
- This function attempts to load five serialized objects from disk using pickle.load:

a. **clf**: The trained classification model (clf.pkl), used to predict the candidate's job role category.
b. **tfidf**: A TfidfVectorizer object (tfidf.pkl), used to vectorize text documents.
c. **le**: A LabelEncoder instance (encoder.pkl), used to convert between category labels and numeric IDs.
d. **ats**: The ATS (Applicant Tracking System) scoring model (ats_scorer.pkl), a regressor that combines similarity metrics into a final score.
e. **prototypes**: A dictionary of prototype profiles (prototypes.pkl), one per job category.

- If all files load successfully, it returns a tuple: (clf, tfidf, le, ats, prototypes).
- If any file is missing (resulting in FileNotFoundError), the function returns (None, None, None, None, None) to indicate resources are unavailable.
- **Caching:**
  The @st.cache_resource decorator ensures these resources are loaded only once per session, improving performance by avoiding repeated disk I/O.
- **Logic Breakdown:**
- **Try Block:**
  - Open and load each required file with pickle.load.
  - If successful, return the loaded objects.
- **Except Block:**
  - If any FileNotFoundError occurs, catch it and return a tuple of Nones.
- **Outcome:** The application checks for None values to determine if the models are available.

Function: clean_text(txt)

Purpose: Preprocess a text string by removing URLs and punctuation, converting to lowercase.

```
def clean_text(txt):
    txt = re.sub(r'http\S+\s', ' ', txt)
    txt = re.sub(r'[^\w\s]', ' ', txt)
    return txt.lower()
```

- **Functionality:**
- Takes a string txt and applies two regular expression substitutions:
    f. **Remove URLs:** re.sub(r'http\S+\s', ' ', txt) replaces any substring starting with "http" up to the next whitespace with a space. This removes web links.
    g. **Remove Punctuation:** re.sub(r'[^\w\s]', ' ', txt) replaces any character that is not a word character (\w) or whitespace (\s) with a space. This strips punctuation and special characters.
- Converts the resulting text to lowercase using txt.lower().
- Returns the cleaned, lowercase string.
- **Logic Breakdown:**
- **URL Removal:** The regex http\S+\s matches any "http" followed by non-space characters and a space, effectively dropping URLs from the text.
- **Punctuation Removal:** The second regex replaces all non-alphanumeric (and non-space) characters with spaces.
- **Lowercasing:** Simplifies downstream text matching by standardizing case.
- **Result:** A normalized string suitable for vectorization or token matching.

Function: extract_text(file)

Purpose: Extract textual content from an uploaded file (PDF, DOCX, or TXT) for analysis.

```
def extract_text(file):
    try:
        if file.name.endswith('.pdf'):
            reader = PyPDF2.PdfReader(file)
            return " ".join([page.extract_text() for page in reader.pages])
```

```
    elif file.name.endswith('.docx'):
        doc = docx.Document(file)
        return " ".join([p.text for p in doc.paragraphs])
    elif file.name.endswith('.txt'):
        return file.read().decode('utf-8')
except:
    return ""
```

- **Functionality:**
- Determines the file type by its extension (.pdf, .docx, or .txt) and extracts text accordingly:
    - **PDFs:** Uses PyPDF2.PdfReader to open the file, then iterates over its pages and extracts text from each page. It concatenates page texts with spaces.
    - **DOCX:** Uses python-docx (imported as docx) to open the Word document, then joins the text of each paragraph with spaces.
    - **TXT:** Reads the uploaded file as bytes and decodes it into a UTF-8 string.
- Returns the full extracted text as a single string.
- If any exception occurs (e.g., unsupported format or read error), returns an empty string.

- **Logic Breakdown:**

- **File Type Checking:**
    - Check if filename ends with .pdf, .docx, or .txt.
- **PDF Extraction:**
    - Initialize PdfReader on the uploaded PDF.
    - For each page object, call extract_text().
    - Join all page texts into one string.
- **DOCX Extraction:**
    - Open the DOCX file as a Document.
    - Iterate through doc.paragraphs and concatenate their .text.
- **TXT Extraction:**
    - Simply read the raw file content and decode to text.
- **Error Handling:**

- Catch exceptions (e.g. file read issues) and return an empty string to signal failure.

Function: calculate_scores(text, category)

Purpose: Compute matching scores between a candidate's resume text and a "master profile" for a predicted job category. Returns an overall ATS score, cosine similarity percentage, and keyword overlap percentage.

```
def calculate_scores(text, category):
    if category not in prototypes:
        return 0, 0, 0

    master_profile = prototypes[category]
    cleaned_resume = clean_text(text)

    # Cosine Similarity
    vecs = tfidf.transform([cleaned_resume, master_profile])
    cosine_sim = cosine_similarity(vecs[0], vecs[1])[0][0]

    # Keyword Match
    res_tokens = set(cleaned_resume.split())
    mp_tokens = set(master_profile.split())
    keyword_match = len(res_tokens.intersection(mp_tokens)) /
len(mp_tokens) if mp_tokens else 0

    # AI Prediction
    try:
        ml_score = ats_model.predict([[cosine_sim, keyword_match]])[0]
    except:
        ml_score = 0

    # Fallback Logic
    if ml_score < 10:
        final_score = cosine_sim * 100
    else:
```

```
        final_score = ml_score
    if final_score < 1:
        final_score *= 100

    return round(final_score, 1), round(cosine_sim*100, 1),
round(keyword_match*100, 1)
```

- **Functionality:**
- **Category Check:** If the predicted category has no prototype in prototypes, immediately return zeros for all scores (no evaluation possible).
- **Master Profile Retrieval:** Fetch the master_profile string for the given category from the prototypes dictionary. Each master profile is a concatenation of cleaned resumes for that role.
- **Text Cleaning:** Use clean_text on the candidate's resume text to standardize formatting.
- **Cosine Similarity Calculation:**
    - Transform the cleaned resume and the master profile into TF-IDF vectors using the loaded tfidf.
    - Compute cosine similarity between the two TF-IDF vectors. This yields a value between 0 and 1 measuring content similarity.
- **Keyword Match Calculation:**
    - Split both cleaned texts into sets of unique tokens (res_tokens and mp_tokens).
    - Compute the intersection size of these token sets.
    - Divide by the number of tokens in the master profile (len(mp_tokens)) to get a keyword overlap ratio.
    - If the master profile has no tokens (unlikely), default overlap to 0.
- **Machine Learning Score Prediction:**
    - Create a feature array [[cosine_sim, keyword_match]].
    - Use the pre-trained ATS scoring model (ats_model) to predict a numeric score (0–100 range) from these features.
    - If prediction fails (caught by except), default ml_score to 0.
- **Fallback Logic for Final Score:**

- If the ML-predicted score (ml_score) is less than 10, use cosine_sim * 100 as the score instead. This accounts for cases when the model underperforms or lacks data.
- Otherwise, use ml_score directly.
- If the resulting final_score is still below 1 (e.g., due to very low similarity), multiply by 100 as a final safety check to avoid near-zero percentages.

- **Formatting:**
  - Return final_score, cosine_sim*100, and keyword_match*100, each rounded to one decimal place. These represent:
  - **ATS Score (%):** The final adjusted match score (higher is better).
  - **Similarity (%):** The raw cosine similarity multiplied by 100.
  - **Overlap (%):** The keyword match ratio multiplied by 100.

Logic Breakdown (Step-by-Step):

- **1. Category Validation:**
  - Check if category exists in the prototypes dictionary.
  - If not present, this implies no reference profile exists; immediately return zero scores. This avoids further errors.
- **2. Prepare Texts:**
  - Retrieve the prototype (master profile text) for the category.
  - Clean the candidate's resume with clean_text.
- **3. Cosine Similarity Computation:**
  - Use the tfidf vectorizer to transform the cleaned resume and master profile into vectors (vecs[0] and vecs[1]).
  - Use cosine_similarity to compute similarity between these two vectors.
  - Extract the scalar similarity value cosine_sim.
- **4. Keyword Overlap Computation:**

    - Create Python set objects of tokens from the cleaned resume and master profile (splitting on whitespace).
    - Compute the intersection of the two sets.
    - Divide the size of the intersection by the total number of tokens in the master profile.
    - This yields the fraction of prototype tokens present in the resume (keyword_match).
- **5. ATS Model Prediction:**
    - Package the two features (cosine similarity and keyword match) into a list-of-list as [[cosine_sim, keyword_match]].
    - Call the ATS regression model's .predict() method with these features.
    - If the prediction throws an error (e.g., model not loaded), handle it by setting ml_score = 0.
- **6. Determine Final Score:**
    - If the machine learning score is suspiciously low (<10), ignore it in favor of the raw cosine similarity percentage. This acts as a fallback heuristic.
    - Otherwise, trust the model's output as the final score.
    - If the resulting score is less than 1 (rare case), multiply by 100 as a catch-all to ensure a reasonable percentage format.
- **7. Output:**
    - Convert similarity and keyword overlap ratios into percentages by multiplying by 100.
    - Round all three numbers to one decimal place for display.
    - Return (ATS_score, similarity_pct, overlap_pct).

Main Application Flow (main())

Purpose: Set up the Streamlit app interface, handle user input, and display results. This function organizes the user experience and orchestrates the use of the above utilities.

```
def main():
   # Header
   st.markdown("<h1> AI Resume Screening</h1>",
```

```
unsafe_allow_html=True)
    st.markdown("<p class='subtitle'>Powered by Machine Learning &
Natural Language Processing</p>", unsafe_allow_html=True)
    if not clf:
        st.error(" Models missing! Run `train_model.py` then
`train_ats_model.py`.")
        st.stop()

    # Upload section
    st.markdown("<br>", unsafe_allow_html=True)
    col1, col2, col3 = st.columns([1, 2, 1])
    with col2:
        file = st.file_uploader(
            "Upload Your Resume",
            type=['pdf', 'docx', 'txt'],
            help="Supported formats: PDF, DOCX, TXT"
        )

    if file:
        # Custom loading animation (omitted for brevity)

        # Extract and process
        text = extract_text(file)

        # Clear loading animation

        if len(text) > 20:
            clean = clean_text(text)
            vec = tfidf.transform([clean])
            cat_id = clf.predict(vec)[0]
            category = le.inverse_transform([cat_id])[0]

            ats_score, raw_sim, key_match = calculate_scores(text,
category)

            st.markdown("<br>", unsafe_allow_html=True)
```

```
        # Category prediction
        st.success(f"###  Predicted Role: **{category}**")

        # Score badge
        if ats_score >= 75:
            badge_class = "score-high"
            emoji = " "
        elif ats_score >= 50:
            badge_class = "score-medium"
            emoji = "  "
        else:
            badge_class = "score-low"
            emoji = "  "

        st.markdown(f"<div class='score-badge {badge_class}'>{emoji}
ATS Score: {ats_score}%</div>", unsafe_allow_html=True)

        # Metrics
        st.markdown("###  Detailed Analysis")
        col1, col2, col3 = st.columns(3)

        with col1:
            st.metric(
                label="  AI Score",
                value=f"{ats_score}%",
                delta="Primary Score"
            )

        with col2:
            st.metric(
                label="  Content Match",
                value=f"{raw_sim}%",
                delta="Similarity"
            )

        with col3:
```

```
            st.metric(
                label="🔑 Keywords",
                value=f"{key_match}%",
                delta="Overlap"
            )

        # Progress bar
        st.markdown("#### Match Strength")
        st.progress(min(ats_score/100, 1.0))

        # Feedback messages
        if ats_score > 75:
            st.balloons()
            st.info("🎉 Excellent match! Your resume aligns well with this role.")
        elif ats_score >= 50:
            st.info("✨ Good match! Consider adding more role-specific keywords to improve.")
        else:
            st.warning("💡 Low match. Try adding more relevant skills and experience keywords.")

        # Extracted text display
        with st.expander("📄 View Extracted Text"):
            st.text_area("Resume Content", text, height=300)
    else:
        st.warning("⚠️ Could not extract text. File might be an image or scan. Please use a text-based document.")
```

This main() function executes when the Streamlit app runs. The overall flow is:

- **Header and Introduction:**
- Display a title and subtitle using Markdown with custom styling (<h1>, <p> tags).
- Check if the models (clf, etc.) are loaded; if not, display an error and stop execution. This ensures the app does not proceed without required resources.
- **File Upload Section:**
- Use st.file_uploader() to allow the user to upload a resume file (.pdf, .docx, or .txt).
- Layout is centered by creating three columns and placing the uploader in the middle column (col2).
- **After File Upload:**
- If a file is uploaded, show a custom loading animation (HTML/CSS) while processing. (The details of this animation are defined in the CSS and included via st.markdown.)
- Call extract_text(file) to read the text content from the uploaded file.
- Once text is extracted, clear the loading animation placeholder.
- **Text Check:**
- If the extracted text length is greater than 20 characters (a sanity check to ensure we got real content), proceed. Otherwise, show a warning about possible failure to extract (e.g., if the resume was an image).
- **Prediction and Scoring:**
- **Clean Text:** Use clean_text(text) to normalize the resume text.
- **Vectorize:** Transform the cleaned resume into a TF-IDF vector (tfidf.transform([clean])).
- **Role Prediction:** Use the classifier clf.predict(vec) to predict a numeric category ID (cat_id). Use the label encoder le to convert this ID back to the category name (e.g., "Data Scientist").
- **Score Calculation:** Call calculate_scores(text, category) to obtain:
    - ats_score: The final ATS match percentage.
    - raw_sim: The raw cosine similarity percentage.

    - key_match: The keyword overlap percentage.
- **Display Results:**
- Insert some spacing via st.markdown("
").
- **Predicted Role:** Show the predicted category in a success message box (st.success).
- **Score Badge:** Determine a CSS class and emoji based on ats_score thresholds (≥75 is high, ≥50 is medium, otherwise low). Display a stylized badge with the score.
- **Detailed Metrics:** Under a "Detailed Analysis" heading, split into three columns to show:
    - **AI Score:** The ATS score percentage.
    - **Content Match:** The cosine similarity percentage.
    - **Keywords:** The overlap percentage.
- **Progress Bar:** Show a progress bar visual with st.progress(ats_score/100), capped at 100%.
- **Feedback Message:** Depending on the ATS score:
    - Score > 75: Celebrate with balloons and show an "Excellent match" info box.
    - 50 ≤ Score ≤ 75: Show a "Good match" info box suggesting improvements.
    - Score < 50: Show a warning encouraging adding more relevant keywords.
- **Resume Content:** Include an expander titled "View Extracted Text" that contains a text area with the full extracted resume text.
- **Error Case:**
- If no valid text was extracted (length ≤ 20), display a warning about potential format issues.

Overall App Structure

- The CSS block at the beginning of app.py (in st.markdown) sets a dark theme, styling headers, containers, metrics, etc. This is UI customization and not core logic, but it ensures a polished look for the app.
- The application makes extensive use of Streamlit components (st.markdown, st.file_uploader, st.columns, st.metric, etc.) to build an interactive interface.

- Modular functions (load_resources, clean_text, extract_text, calculate_scores) separate concerns:
- load_resources handles model loading.
- clean_text standardizes text.
- extract_text reads resume content.
- calculate_scores computes evaluation metrics.
- The main() function then ties everything together to provide a complete user experience, from uploading a resume to seeing a detailed analysis.

Classifier Training (train_classifier.py)

The train_classifier.py script trains the classification model used by app.py. It includes data loading, cleaning, prototype creation, vectorization, model training, and saving the trained objects. The core function is train_classifier().

Function: train_classifier()

Purpose: Load a resume dataset, preprocess it, build prototypes, train a multi-label classifier to predict job roles, and save the resulting models.

```
def train_classifier():
    # 1. Load Dataset (AzharAli05)
    print("Loading AzharAli05/Resume-Screening-Dataset...")
    try:
        ds = load_dataset("AzharAli05/Resume-Screening-Dataset")
        df = pd.DataFrame(ds['train'])
        print(f"Loaded {len(df)} resumes.")
    except Exception as e:
        print(f"Error loading dataset: {e}")
        exit()
```

- **Dataset Loading:**
- Uses the datasets library to fetch the "AzharAli05/Resume-Screening-Dataset".
- Converts the training portion of the dataset into a Pandas DataFrame df.
- Reports the number of resumes loaded.
- If loading fails, prints an error and exits.

```
    # 2. Setup Columns
    text_col = 'Resume'
    label_col = 'Role'
```

- **Column Definition:**

- Defines which DataFrame columns contain the resume text ('Resume') and the corresponding label ('Role').
- These should match the dataset's structure.

```
    # 3. Cleaning Function
    def clean_resume(txt):
        cleanText = re.sub(r'http\S+\s', ' ', str(txt))
```

```
        cleanText = re.sub(r'RT|cc', ' ', cleanText)
        cleanText = re.sub(r'#\S+\s', ' ', cleanText)
        cleanText = re.sub(r'@\S+', '  ', cleanText)
        cleanText = re.sub(r'[!"#$%&\'()*+,-./:;<=>?@[\]^_`{|}~]', ' ',
cleanText)
        cleanText = re.sub(r'[^\x00-\x7f]', ' ', cleanText)
        cleanText = re.sub(r'\s+', ' ', cleanText)
        return cleanText

    print("Cleaning data...")
    df['cleaned_resume'] = df[text_col].apply(clean_resume)
```

- **Data Cleaning:**

- Defines an inner function clean_resume(txt) that takes a text string and applies several regex substitutions:
  - Remove URLs (http...).
  - Remove Twitter-like indicators (RT, cc).
  - Remove hashtags (#tag) and mentions (@user).
  - Remove punctuation characters explicitly.
  - Remove non-ASCII characters.
  - Collapse multiple whitespace into single spaces.
- Applies this cleaning function to every resume text in the DataFrame, storing results in a new column 'cleaned_resume'.
- Prints a message indicating that cleaning is happening.

```
    # 4. Generate & Save Prototypes (Crucial for App)
    print("Generating Master Profiles (Prototypes)...")
    prototypes = df.groupby(label_col)['cleaned_resume'].apply(lambda
x: ' '.join(x)).to_dict()
    pickle.dump(prototypes, open('prototypes.pkl', 'wb'))
```

Prototype Creation:

- Groups the DataFrame by the job role label (Role).
- For each role, concatenates all cleaned resumes into one long string. This aggregated string is a “master profile” for that role.
- Stores these master profiles in a dictionary prototypes where keys are role names and values are combined text.
- Serializes (pickle.dump) the prototypes dict to prototypes.pkl for later use in scoring.
- Prints a message to indicate this generation step.

```
# 5. Encoding Labels
le = LabelEncoder()
df['Category_ID'] = le.fit_transform(df[label_col])
```

Label Encoding:

- Initializes a LabelEncoder to convert categorical role names into numeric IDs.
- Applies fit_transform on the 'Role' column to produce a new integer column 'Category_ID'.
- This maps each unique role to a unique integer (0,1,2,...).
- The encoder le will be saved later for reversing this mapping.

```
# 6. Vectorizing
print("Vectorizing...")
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
tfidf.fit(df['cleaned_resume'])
requiredText = tfidf.transform(df['cleaned_resume'])
```

TF-IDF Vectorization:

- Instantiates a TfidfVectorizer with English stop words removed and a maximum of 5000 features. TF-IDF converts text into numerical vectors.

- Calls fit on all cleaned resumes, learning the vocabulary and IDF values from the entire corpus.
- Transforms the cleaned resumes into a matrix requiredText of TF-IDF features (rows correspond to resumes).
- Prints "Vectorizing..." to indicate progress.

```
# 7. Training
print("Training Classifier...")
clf = OneVsRestClassifier(KNeighborsClassifier())
clf.fit(requiredText, df['Category_ID'])
```

Classifier Training:

- Creates an OneVsRestClassifier wrapping a KNeighborsClassifier. This multi-label setup means it can predict multiple roles (though the dataset might be single-label per resume).
- Fits this classifier on the TF-IDF vectors (requiredText) and the numeric category IDs (Category_ID).
- After training, clf can take a TF-IDF vector and predict which role(s) it belongs to.
- Prints a status message for training.

```
# 8. Saving Models
print("Saving models...")
pickle.dump(clf, open('clf.pkl', 'wb'))
pickle.dump(tfidf, open('tfidf.pkl', 'wb'))
pickle.dump(le, open('encoder.pkl', 'wb'))
print("SUCCESS: Classification models + Prototypes saved.")
```

Saving Trained Objects:

- Serializes and saves:
  - The trained classifier model clf to clf.pkl.
  - The fitted TF-IDF vectorizer to tfidf.pkl.
  - The label encoder le to encoder.pkl.
- Confirms success with a printed message.

Logic Breakdown (Step-by-Step):

- **Data Loading:** Import the training data from an online source (Hugging Face dataset). Handle potential errors.
- **Data Preparation:** Identify which columns in the data contain text and labels.
- **Text Cleaning:** Define and apply a regex-based cleaning function to remove unwanted characters and formatting from resumes. This results in cleaned_resume.
- **Prototype Generation:** Group by job role and concatenate all cleaned resumes for each role into a single "master profile." These prototypes serve as reference profiles for scoring later. Save them for the application.
- **Label Encoding:** Convert categorical role labels into numeric form (Category_ID) using LabelEncoder.
- **TF-IDF Vectorization:** Initialize a TF-IDF vectorizer (with a defined maximum feature count). Fit it on the cleaned texts to learn vocabulary and IDF values. Transform all texts into TF-IDF feature vectors.
- **Model Training:** Create a multi-label classifier (OneVsRest with KNN) and fit it on the TF-IDF vectors and category IDs.
- **Saving Outputs:** Serialize the trained classifier, TF-IDF vectorizer, and label encoder to disk. These files (clf.pkl, tfidf.pkl, encoder.pkl) will be loaded by the app for predictions.

Notes:

- **OneVsRest with KNN:** Since resumes may have multiple applicable roles, a One-Vs-Rest strategy allows the KNN to handle multi-label cases (one KNN model per role).
- **Prototypes:** These are crucial for the ATS scoring phase to compute similarities against each role's collective profile.
- The entire function reports progress via print statements for traceability.

ATS Scorer Training (train_ats_scorer.py)

The train_ats_scorer.py script creates the regression model used to compute the final ATS match score from similarity metrics. It sets up synthetic training data for demonstration purposes and trains a Gradient Boosting Regressor.

Function: train_ats_scorer()

Purpose: Train a regression model that predicts an ATS match percentage from cosine similarity and keyword overlap features, then save the model.

```
def train_ats_scorer():
    print("☑ Loading TF-IDF Vectorizer...")
    try:
        tfidf = pickle.load(open("tfidf.pkl", "rb"))
    except FileNotFoundError:
        print("❌ ERROR: tfidf.pkl not found. Run train_model.py first.")
        return
```

- **TF-IDF Check:**
- Attempts to load the previously saved TF-IDF vectorizer from tfidf.pkl. This ensures the same text vectorization is used.
- If the file is missing, it prints an error and exits.

```
    print("☑ Generating ATS training data...")
```

```
X = np.array([
    [0.95, 0.90],
    [0.90, 0.85],
    [0.80, 0.70],
    [0.70, 0.60],
    [0.60, 0.50],
    [0.45, 0.35],
    [0.30, 0.20],
    [0.20, 0.10]
])

y = np.array([98, 92, 82, 70, 60, 45, 30, 20])
```

Synthetic Training Data:

- Manually defines a small set of example feature vectors X and target scores y. Each row in X represents [cosine_similarity, keyword_match] values (normalized between 0 and 1).
- The corresponding y array holds handcrafted ATS score percentages (0-100).
- These synthetic points represent plausible combinations of similarity and overlap mapping to final scores. For example:
    - If cosine similarity ≈ 0.95 and keyword match ≈ 0.90, the final ATS score is set to 98.
    - If similarity 0.20 and match 0.10, score is 20.
- This is a simplified demonstration dataset for training.

```
print("☑ Training Gradient Boosting ATS model...")
model = GradientBoostingRegressor(
    n_estimators=100,
    learning_rate=0.1,
    max_depth=3,
    random_state=42
)
model.fit(X, y)
```

Model Training:

- Instantiates a GradientBoostingRegressor with:
  - 100 boosting stages (n_estimators=100).
  - A learning rate of 0.1.
  - Maximum tree depth of 3 for each weak learner.
  - A fixed random_state for reproducibility.
- Fits the regressor on the synthetic features X and targets y.
- The model learns to predict a score given any new pair of similarity features.

```
with open("ats_scorer.pkl", "wb") as f:
    pickle.dump(model, f)
print("🎉 SUCCESS: ats_scorer.pkl created!")
```

Saving the Model:

- Serializes the trained model to ats_scorer.pkl.
- Confirms success with a printed message.

Logic Breakdown (Step-by-Step):

- **TF-IDF Vectorizer Validation:**
  - Load the TF-IDF vectorizer. It ensures consistency, though this script doesn't actually use tfidf beyond this check.

- **Generate Training Data:**
  - Define example feature vectors (X) representing different levels of cosine similarity and keyword overlap.
  - Define target ATS scores (y) that correspond to each example pair. These reflect intuitive relationships (higher similarity → higher score).

- **Train Regression Model:**
    - Initialize a GradientBoostingRegressor with specified hyperparameters.
    - Fit the model to the synthetic dataset. The gradient boosting regressor will learn a smooth mapping from (similarity, overlap) → ATS score.

- **Save the ATS Model:**
    - Use pickle.dump to save the fitted model as ats_scorer.pkl. This model will be loaded by the main app to predict final scores.

- **Status Reporting:**
    - Print messages at each major step to indicate progress and success.

Notes:

- The training set here is handcrafted and small; in a production scenario, one might generate or gather more extensive data. However, this serves to illustrate the workflow.
- The choice of GradientBoostingRegressor provides a nonlinear regression capable of modeling complex relationships between inputs and output.
- The script does not explicitly return the model; saving it to a file is the goal.

Chapter 5: Scope and Limitations

Introduction to Operational Boundaries

In the rigorous discipline of software engineering, particularly within the specialized sub-field of Applied Artificial Intelligence (AI), the definition of scope serves as the fundamental contract between the system's design and its deployment reality. It is insufficient to merely list the features a system possesses; rather, one must rigorously delineate the **Operational Design Domain (ODD)**. This concept, borrowed from autonomous systems engineering, refers to the specific conditions—environmental, varying data inputs, and interaction constraints—under which a system allows for reliable function.

The Applicant Tracking System (ATS) developed in this research is not designed as a universal text comprehension engine, nor is it a general-purpose artificial intelligence capable of open-ended reasoning or "Zero-Shot" generalization. Instead, it is a specialized **Decision Support System (DSS)**, engineered to operate within a specific, controlled set of parameters. This chapter establishes the "Contract of Reliability" for the system. It asserts that the high performance metrics (e.g., the 98% classification accuracy cited in Chapter 4) are valid *if and only if* the input data adheres to the constraints defined herein.

By explicitly defining these boundaries—ranging from the byte-level encoding of input files to the mathematical assumptions of the inference engine—we distinguish between the system's intended functionality and its indeterminate behavior in Out-of-Distribution (OOD) scenarios. This distinction is critical for minimizing the risk of "AI Hallucination" or confident misclassification, ensuring that human operators understand exactly where the machine's competence ends and where human oversight must begin.

Domain Specificity: The Target Industry Vertical
The first and most critical dimension of scope is the **Domain Specificity**. While the underlying mathematical principles of the Vector Space Model (VSM) and K-Nearest Neighbors (KNN) classification are theoretically domain-agnostic, the *semantic intelligence* of this specific implementation is strictly bound by the data it has ingested. The "Master Profiles" (centroid vectors) are artifacts of specific training data, and thus the system's "worldview" is limited to the industry represented in that data.
Primary Scope: Information Technology (IT)
The developed system is primarily scoped, optimized, and validated for the **Information Technology (IT)** and **Software Engineering** sectors. This restriction is not arbitrary; it is a deliberate design choice driven by the favorable linguistic characteristics of technical resumes, which align specifically well with the statistical assumptions of Bag-of-Words (BoW) models.
1. The "High-Signal" Vocabulary of Tech Technical resumes are characterized by a uniquely high density of **Proper Nouns** and **Standardized Named Entities**.

- **Orthogonality in Vector Space:** Terms such as "Python," "AWS," "Kubernetes," "React," and "Docker" act as strong, orthogonal features in a high-dimensional vector space. They are distinct, binary indicators of competence. A candidate either lists "Docker" or they do not.
- **Contrast with Soft-Skill Domains:** In contrast, resumes in domains such as "Sales," "Executive Management," or "Creative Writing" rely heavily on abstract verbs and narrative flow (e.g., "Led," "Optimized," "Synergized," "Spearheaded"). These terms are highly polysemous (having multiple meanings) and context-dependent. "Optimized" could refer to a database query (in IT) or a supply chain route (in Logistics). Because TF-IDF relies on term frequency rather than deep contextual understanding, it struggles to differentiate these nuances.
- **Design Justification:** By restricting the scope to IT, we maximize the signal-to-noise ratio. The presence of specific technical keywords correlates strongly with role suitability, allowing the shallow learning algorithms (KNN/Gradient Boosting) to perform with high precision without requiring deep semantic transformers.

2. Global Standardization of Nomenclature The IT industry possesses a globally standardized taxonomy that is rare in other fields.

- **Universal Lexicon:** A "DevOps Engineer" in Bangalore performs largely the same tasks using the same toolchain (Linux, Jenkins, Git) as a DevOps Engineer in San Francisco. The vocabulary does not shift significantly based on geography.
- **Robustness of Prototypes:** This standardization ensures that the "Master Prototypes" generated by the system are robust and universally applicable. A prototype built on US-based resumes will still effectively score an India-

based resume because the core entities (e.g., "Java," "SQL") remain identical. This allows the system to scale across geographic boundaries within the strict vertical of technology.

Validated Job Clusters

The system is explicitly validated to process and evaluate resumes mapping to the specific high-dimensional clusters found in the *AzharAli05* training dataset. The operational scope is strictly limited to the following categories:

- **Data Science & Machine Learning:**
  - *Scope:* The system recognizes the specific lexicon of this domain, including programming languages (Python, R), libraries (Pandas, Scikit-learn, TensorFlow, PyTorch), and theoretical concepts (Regression, Clustering, Neural Networks).
  - *Boundary:* It is optimized for "Applied Data Science" (using tools) rather than "Theoretical Research" (writing proofs), as the latter often lacks standard keyword density.
- **Web Development (Full Stack):**
  - *Scope:* The system covers the entire stack. Frontend terminologies (HTML, CSS, JavaScript, React, Vue, Angular) and Backend terminologies (Node.js, Django, Flask, PHP) are within scope.
  - *Boundary:* It assumes modern web architectures. Legacy web technologies (e.g., Flash, Silverlight) may be underrepresented in the training data, leading to lower scoring accuracy for legacy roles.
- **Database Administration:**
  - *Scope:* The system is calibrated to evaluate competency in Relational Database Management Systems (SQL, PostgreSQL, MySQL, Oracle) and NoSQL technologies (MongoDB, Cassandra, Redis).
  - *Boundary:* It focuses on database *management* and *querying*. It may be less effective at scoring hardware-level storage engineering.
- **Network Engineering & Security:**
  - *Scope:* The scope includes Infrastructure-as-Code (Terraform, Ansible), cybersecurity protocols (SSL/TLS, Firewalls), and hardware configuration (Cisco/Juniper routing).
  - *Boundary:* It distinguishes between "Network Administration" (maintenance) and "Network Architecture" (design) primarily through keyword frequency.
- **Project Management & HR:**
  - *Scope:* As a control group present in the training data, the system can also evaluate HR-specific resumes (Keywords: "Recruitment," "Onboarding," "Employee Relations") and Technical Project Management (Keywords: "Agile," "Scrum," "JIRA," "Waterfall").
  - *Boundary:* This serves as the boundary for non-technical evaluation.

Out-of-Distribution (OOD) Exclusion: Resumes falling outside these specific clusters—such as those for **Medical Practitioners** (Nurses, Doctors), **Civil**

Engineers, **Legal Consultants**, or **Creative Artists**—are considered Out-of-Distribution.

- *The Force-Fit Error:* If a user uploads a "Medical Nurse" resume, the KNN classifier will attempt to force-fit the document into the nearest available technical cluster based on shared stop-words or generic soft skills. For example, it might classify a Nurse as an "HR Manager" because both resumes contain words like "People," "Care," and "Management."
- *Invalidity:* The resulting classification and suitability score will be statistically invalid. Therefore, the evaluation of non-technical resumes falls strictly outside the guaranteed operational scope.

Linguistic Scope: The Monolingual Architecture

The linguistic scope defines the boundaries of the system regarding Natural Language Processing (NLP). Unlike human recruiters who are often multilingual or capable of using translation tools, this system is architected as a **Monolingual English Processing Engine**. This constraint is not merely a preference but a hard technical dependency hard-coded into the preprocessing pipeline.

Language Constraints

The system is engineered, trained, and validated **exclusively for the English language**.

1. Stop-Word Dependency The text preprocessing module relies on the NLTK (Natural Language Toolkit) standard English stop-word list, or the built-in list within scikit-learn.

- *Mechanism:* The TfidfVectorizer is initialized with stop_words='english'. This tells the vectorizer to automatically filter out high-frequency, low-semantic-value English tokens such as "the," "and," "is," "for," "which," and "on."
- *The Failure Mode:* The system contains no logic for multi-lingual stop-word detection or language identification (LID). If a **French** resume is input:
    - Common terms like "le," "la," "et," "pour," and "avec" will *not* be removed.
    - Because these words appear with extremely high frequency in the document, the TF-IDF algorithm (Term Frequency) will assign them significant weight.
    - Since these words do not appear in the (English) training corpus, they will be treated as unique, rare keywords.
    - *Result:* Massive noise injection into the vector space, rendering the Similarity Score (Cosine = 0.0) and the Classification invalid.

2. Character Encoding and Script Support The extraction and cleaning modules are optimized for **UTF-8** encoding but specifically tuned for the **Latin Alphabet (ASCII/ANSI)**.

- *Regex Scope:* The regular expression used for cleaning—re.sub(r'[^\w\s]', ...)—is designed to strip non-alphanumeric characters. In the Python re library, \w matches [a-zA-Z0-9_].

- *Script Exclusion:*
    - **Logographic Scripts:** Chinese (Hanzi), Japanese (Kanji/Kana). While UTF-8 supports these, the regex cleaning may fragment them or the tokenizer (splitting by whitespace) will fail, as these languages often do not use whitespace delimiters.
    - **Right-to-Left Scripts:** Arabic, Hebrew, Urdu. The system logic assumes Left-to-Right reading order. Bi-directional text (mixing English and Arabic) will cause extraction artifacts where sentences are scrambled.
    - **Non-Latin Alphabets:** Cyrillic (Russian), Greek, Hindi (Devanagari). These are explicitly outside the primary scope.

The Lexicon Boundary (Vocabulary Scope)

The system's "vocabulary"—the universe of words it recognizes—is bounded by the **Training Corpus Distribution**.

1. Feature Dimensionality Limit The TfidfVectorizer is configured with max_features=5000.

- *Mechanism:* The vectorizer ranks every unique word in the 10,000+ training resumes by frequency. It selects the top 5,000 most frequent words to build the vocabulary dictionary. All other words are discarded.
- *Operational Scope:* The system only "knows" and monitors these top 5,000 words.
- *The "Long Tail" Exclusion:* Any technical term, slang, proper noun, or specific company name that falls into the "Long Tail" (i.e., it is rare and falls outside the top 5,000) is mathematically invisible to the system.
    - *Example:* If "Generative Adversarial Networks" appears only once in the training data, it may be dropped. If a candidate uses this term, the system will not register it. This restricts the scope of evaluation to **Standard, Widely-Used Industry Terminology**.

2. Case Sensitivity and Normalization The scope includes a strict normalization protocol: **Lowercasing**.

- *Mechanism:* clean_text(txt).lower().
- *Implication:* The system treats "PYTHON," "Python," and "python" as identical tokens.
- *Limitation:* This destroys distinctions where case matters. For example, "Go" (the language, usually capitalized) vs "go" (the verb). The system flattens these into a single token "go," relying on Inverse Document Frequency (IDF) to dampen the common verb usage. This creates a scope limitation where ambiguous capitalization can introduce noise.

Data Ingestion Scope: The File Parsing Pipeline

The Input/Output (I/O) scope is defined by the capabilities of the text extraction module. The system is designed to function as a **Linearization Pipeline**, converting complex, two-dimensional document formats into normalized, one-dimensional character streams.

File Format Compatibility

The system is explicitly scoped to process **Digital-Native Documents**. This refers to files generated directly from word processing software where text exists as encoded character data.

1. Portable Document Format (.pdf) The system utilizes the PyPDF2 library for PDF ingestion.

- *Version Scope:* It supports standard PDF versions 1.4 through 1.7.
- *Stream Logic:* The system parses the ContentStream of each page object. It assumes a standard character encoding (WinAnsiEncoding or UTF-8).
- *Exclusions:*
    - **Encrypted PDFs:** Files protected by user passwords are out of scope; the system will fail to open them.
    - **Rasterized (Scanned) PDFs:** PDFs that consist solely of images (e.g., a photo of a paper resume converted to PDF) are out of scope. PyPDF2 cannot extract text from images. The system will return an empty string for these files.

2. OpenXML Documents (.docx) The system interfaces with the XML hierarchy of modern Microsoft Word documents using python-docx.

- *Scope of Extraction:* The system iterates through the <w:p> (paragraph) elements of the document.xml structure.
- *Granularity:* It preserves paragraph breaks as logical delimiters but explicitly discards rich-text formatting tags (bold, italics, underline, color, font size). The operational scope focuses purely on the *semantic payload* of the text, ignoring the *presentation layer*.
- *Legacy Exclusion:* Binary .doc files (pre-Office 2007) are outside the scope, as they require complex OLE2 binary parsing which the current library stack does not support.

3. Plain Text (.txt) This format serves as the operational baseline for unformatted data ingestion. The scope assumes **UTF-8** encoding.

The "Linearization" Assumption

A critical scope definition is the treatment of document geometry. The system is scoped to perform **Linear Extraction**.

- *The Assumption:* The system assumes that the logical reading order of the document matches the internal creation order of the data stream.
- *Layout Abstraction:* Complex visual hierarchies—such as text boxes, floating sidebars, headers, and footnotes—are flattened into a single string.
- *The "Reading Order" Hazard:* In complex multi-column layouts (e.g., "Skills" on the left, "Experience" on the right), the extractor may read across the page, merging unrelated text blocks. The scope does not include **Document Layout Analysis (DLA)**. The system does not "know" that text on the left is a sidebar; it processes the document as a continuous sequence of words. This limits the scope to resumes where the semantic meaning is preserved even when the visual structure is removed.

Algorithmic Scope: The Inference Engine
The "Algorithmic Scope" defines the specific type of machine intelligence employed by the system. Unlike general-purpose AI which attempts to model cognitive reasoning, this system is scoped as a **Statistical Pattern Matching Engine**. It utilizes **Shallow Learning** and **Instance-Based Learning** paradigms, explicitly excluding Deep Learning or Generative AI (Large Language Models) from its core logic. This choice dictates the boundaries of its reasoning capabilities.

The "Frozen Intelligence" Paradigm

The system operates as a **Static Inference Engine**.

- **Offline Training:** The intelligence of the system is derived entirely from the offline training phase. The artifacts generated—the Classifier (clf.pkl), the Scorer (ats_scorer.pkl), and the Master Prototypes (prototypes.pkl)—are static.
- **No Online Learning:** The scope does not include "Online Learning" or "Active Learning." The system does not update its weights, expand its vocabulary, or refine its prototypes based on user interactions in real-time. If a user corrects a misclassification, the system does not "learn" from this correction for future sessions. The intelligence is effectively "frozen" at the moment of deployment.
- **Implication:** This restricts the scope to **Stationary Environments**. The system assumes that the definition of a "Data Scientist" today is roughly the same as it was when the training data was collected. It cannot dynamically adapt to rapid shifts in industry terminology without a manual retraining cycle.

Classification Strategy: Instance-Based Logic

The scope of categorization is defined by the **K-Nearest Neighbors (KNN)** algorithm using a **One-vs-Rest (OvR)** strategy.

1. Instance-Based Learning

KNN is a "lazy learner." It does not abstract the training data into rules (like a Decision Tree) or probability distributions (like Naive Bayes). Instead, it memorizes the geometric distribution of the training vectors.

- **Scope of Inference:** The system determines the role of a new resume solely by calculating its Euclidean distance to existing resumes in the 45-dimensional training space.
- **Boundary:** The scope is limited to the **density of the training clusters**. If the training data for "Network Engineer" is sparse or scattered, the classification boundary will be ill-defined, leading to lower confidence predictions.

2. Closed-Set Classification

The system is scoped to classify resumes into one of the 45 pre-defined categories present in the AzharAli05 dataset.

- **The "None-of-the-Above" Limit:** The system lacks a mechanism to detect "Unknown" or "New" categories. It operates on a **Closed-Set Assumption**.
- **Failure Mode:** If a completely novel role emerges (e.g., "Quantum Algorithmist"), the system cannot label it as "Unknown." It is mathematically forced to classify it into the nearest existing neighbor (e.g., "Data Scientist"). The system cannot dynamically discover new categories zero-shot.

Scoring Logic: Hybrid Feature Engineering

The scope of evaluation is defined by a **Hybrid Regression Model** utilizing Gradient Boosting.

1. Feature Scope

The system scores candidates based on exactly two derived features:

- **Semantic Similarity:** The Cosine Distance between the candidate's vector and the Master Profile vector. This measures "Content Relatedness."
- **Lexical Overlap:** The Jaccard-like intersection of token sets. This measures "Vocabulary Precision."
- **Exclusion of Metadata:** The system does not evaluate metadata features. It ignores "Years of Experience" (temporal extraction), "University Ranking" (entity recognition), "Candidate Location" (geo-filtering), or "Sentiment" (tone analysis). The score is strictly a measure of *textual similarity*, not holistic employability.

2. The Deterministic Fallback

The scope includes a heuristic boundary for scoring reliability.

- **Mechanism:** The code implements a fallback: if ml_score < 10: final_score = cosine_sim * 100.
- **Rationale:** This limits the scope of the Machine Learning model's authority. If the AI model predicts a catastrophic failure (score near 0) but the semantic similarity is high, the system overrides the AI in favor of the deterministic math. This ensures that the system remains "safe" and predictable even when the ML model encounters an outlier.

Deployment Architecture: The "Stateless Prototype"

The deployment scope is strictly defined as a **Functional Prototype** utilizing the **Streamlit** framework. It is engineered for demonstration, pilot testing, and algorithmic validation, rather than as a commercial multi-tenant SaaS platform.

5.6.1 Stateless Execution Model

The system operates on a **Reactive Execution Model**, which fundamentally differs from traditional stateful web applications.

1. Transient Memory State

The system operates without a persistent backend database (SQL or NoSQL).

- **Session Scope:** All candidate data—the uploaded file object, the extracted text string, the calculated vectors, and the final scores—exists only in the server's **Random Access Memory (RAM)**.
- **Lifecycle:** This data persists only for the duration of the active browser session. Once the user closes the tab, refreshes the page, or if the Streamlit server restarts, the data is instantly flushed (Garbage Collected).
- **Privacy Implication:** This creates a privacy-by-design scope where no personal data is stored on disk (other than the temporary buffer during upload). However, it also means the system cannot perform "Longitudinal Analysis" (tracking a candidate over time).

2. Atomic Processing

The system is scoped for Atomic Transactions.

- **Mechanism:** It processes one resume at a time.
- **Boundary:** It does not support "Batch Processing" (e.g., uploading a ZIP file containing 1,000 resumes). The algorithmic scope is focused on the deep analysis of a *single* instance, providing immediate feedback to a human operator, rather than high-throughput batch filtering.

Computational Envelope

The system is scoped for **Local or Single-Instance** execution.

1. CPU-Bound Architecture

The system relies entirely on the Central Processing Unit (CPU) for operation.

- **Tasks:** Text parsing (Regex), Vectorization (Sparse Matrix Multiplication), and Inference (Tree Traversal).
- **Hardware Scope:** It does not require Graphics Processing Unit (GPU) acceleration. This distinguishes it from Deep Learning models (like BERT or GPT), making it deployable on standard commodity hardware (e.g., a standard laptop or an AWS t2.micro instance).

2. Concurrency Limits

The application is single-threaded in its request handling (inherent to the standard Streamlit deployment model).

- **Throughput:** It is designed to handle a single user or a very low volume of concurrent users.

- **Exclusion:** It does not include load balancers, container orchestration (Kubernetes), or asynchronous task queues (Celery/Redis) necessary for handling thousands of simultaneous requests.

Exclusionary Scope (What is Out of Scope)

A rigorous engineering definition must explicitly state what the system *cannot* do. The following capabilities are explicitly excluded from the current project scope to manage complexity and computational cost.

Optical Character Recognition (OCR)

The system is strictly a **Text-Processing Pipeline**, not an **Image-Processing Pipeline**.

- **Analog Data:** The system cannot process analog or rasterized data.
- **Scanned Resumes:** A resume submitted as a flattened image (JPEG, PNG) or a PDF containing only images (e.g., a photo of a paper resume) is outside the scope. The PyPDF2 library requires a text layer to function.
- **Rationale:** Integrating OCR (e.g., Tesseract or AWS Textract) would introduce significant latency (seconds per page) and require heavy dependencies, violating the "lightweight prototype" design constraint.

Generative Capabilities

The system is a **Discriminator**, not a **Generator**.

- **No Rewrite Logic:** The system cannot rewrite a candidate's resume to improve it or suggest better phrasing.
- **No Natural Language Feedback:** While it provides numeric scores and metric breakdowns, it does not generate qualitative feedback paragraphs (e.g., "You should add more SQL keywords to your summary"). This limitation is consistent with the decision to use Statistical ML rather than Generative LLMs.

Verification and Background Checking
The system evaluates **Content**, not **Veracity**.

- **Truthfulness:** The system operates on the assumption that the input data is honest. It cannot cross-reference a candidate's claims with external databases (e.g., LinkedIn API, University Registrars, Criminal Background Checks).
- **Deep Verification:** Verifying whether a candidate *actually* possesses a degree or *actually* worked at a specific company is outside the algorithmic scope. The system measures the *semantic signal* of the claim, not the *truth* of the claim.

In summary, the developed Applicant Tracking System is defined by a specific set of operational boundaries. It is an **English-language, text-based, IT-optimized, stateless decision support tool**. It excels at the high-speed pattern matching of technical resumes against historical baselines, provided the input data is digital-native and semantically standard. It is not a replacement for human judgment, nor is it a universal text comprehension engine. These boundaries define the "Safe Operating Area" for the technology, ensuring that users understand both its power and its requisite constraints.

Limitations and Constraints

Introduction to Systemic Limitations

In the rigorous evaluation of any engineering artifact, particularly those governed by probabilistic logic such as Machine Learning, the definition of limitations is as critical as the definition of capabilities. While the proposed Applicant Tracking System (ATS) demonstrates high efficacy within its defined scope—achieving robust classification accuracy on the test set—it is not an infallible oracle. It is a statistical model approximating human judgment, and like all models, it is subject to error, bias, and blindness.

This section provides a critical, structural analysis of the system's limitations. Unlike "bugs" which can be fixed with code patches, these are **Epistemological** and **Structural Constraints**—fundamental boundaries imposed by the choice of architecture (Vector Space Models) and the nature of the data itself. We categorize these constraints into four distinct domains: **The Semantic Gap** (linguistic limitations), **The Structural Gap** (parsing limitations), **The Algorithmic Gap** (mathematical limitations), and **The Veracity Gap** (ethical and data limitations).

The Semantic Gap: Statistical Correlation vs. Cognitive Understanding

The most profound limitation of the developed system lies in the fundamental difference between *processing text* and *understanding language*. The system utilizes a **Vector Space Model (VSM)** architecture, specifically **TF-IDF (Term Frequency-Inverse Document Frequency)** combined with **Cosine Similarity**. While mathematically robust, this approach suffers from the "Semantic Gap."

The Bag-of-Words (BoW) Blindness

The TF-IDF vectorizer operates on a "Bag-of-Words" assumption, which posits that the order of words in a document does not matter, only their frequency.

- **Loss of Syntactic Structure:** The system decomposes a resume into a pile of independent tokens. It destroys the grammatical structure that gives sentences meaning.
- **The "Agency" Failure Mode:**
    - *Example:* Consider the sentence *"I managed the project lead"* versus *"I was managed by the project lead."*
    - *System Interpretation:* To a Bag-of-Words model, these sentences are nearly identical because they contain the same set of high-value keywords ("managed," "project," "lead"). The preposition "by" is often filtered out as a stop-word.
    - *Consequence:* The system fails to distinguish between the **Subject** (the doer) and the **Object** (the receiver). Consequently, it may credit a candidate for leadership skills they do not possess, simply because they used leadership-related verbs in a passive context.

Inability to Process Negation and Conditionals

Due to the reliance on n-grams (mostly unigrams and bigrams), the system struggles to accurately interpret negation or complex conditional logic.

- **The Negation Failure Mode:**
  - *Scenario:* A candidate writes *"I have no experience in Java, but I am willing to learn."*
  - *System Interpretation:* The TF-IDF vectorizer extracts "Java" as a high-value feature. The negator "no" is often filtered out as a stop-word or carries insufficient weight to counteract the strong signal of the technical keyword.
  - *Consequence:* The system scores this candidate highly for Java competency, a **False Positive** resulting from statistical blindness.
- **The Conditional Failure Mode:**
  - *Scenario:* A candidate writes *"I aimed to design a system using Kubernetes"* (implying the design never happened) versus *"I designed a system using Kubernetes."*
  - *Consequence:* The system treats hypothetical intent and historical action identically. It scores the *mention* of the technology, not the *utilization* of it.

The Polysemy and Homonymy Challenge

In a high-dimensional vector space, each dimension represents a specific token. The system assumes a one-to-one mapping between a word and a concept. Natural language, however, violates this assumption through **Polysemy** (one word, multiple meanings).

- **Ambiguity in Technical Lexicons:**
  - The word **"Go"** can refer to the programming language (Golang) or the common verb (e.g., "Ready to go").
  - The word **"React"** can refer to the JavaScript framework or a behavioral verb (e.g., "Reacted to incidents").
  - The word **"Code"** can refer to software programming or building regulations (e.g., in a Civil Engineering resume).
- **Impact:** Although **Inverse Document Frequency (IDF)** attempts to dampen the weight of common words, it cannot distinguish context. A non-technical resume that uses the word "express" frequently (e.g., "I express my ideas clearly") might be falsely vectorized near a "Node.js Developer" prototype because "Express.js" is a key backend technology. This introduces noise into the similarity calculation, potentially inflating scores for irrelevant candidates.

The Structural Gap: Layout Sensitivity and Parsing Fragility

The system acts as a **Linearization Pipeline**, converting 2D documents (PDFs) into 1D strings. This transformation is lossy. The visual layout of a resume carries significant information that is often discarded or corrupted during the extraction process.

The "Reading Order" Hazard in Multi-Column Layouts
Modern resumes frequently utilize complex, magazine-style layouts with multiple vertical columns (e.g., a "Skills" sidebar on the left and an "Experience" main body on the right).

- **The Parsing Error:** The PyPDF2 library parses text streams based on the internal creation order of the PDF elements, not their visual position. In many cases, it acts as a "horizontal scanner," reading across the entire width of the page.
- **Data Corruption:** This results in the interleaving of disjointed text blocks. A sentence in the main column ending with *"Responsible for managing..."* might be instantly followed by a bullet point from the sidebar like *"Python, SQL,"* resulting in the nonsensical string: *"Responsible for managing Python, SQL."*
- **Consequence:** This destruction of local context prevents the model from accurately associating specific skills with specific job roles or time periods. The n-gram context is polluted, degrading the precision of the similarity measure.

Loss of Temporal Context (Chronology)
Because the system aggregates the entire resume into a single global vector (master_string), it loses the temporal dimension of the candidate's career.

- **The "Flattening" Effect:** The system cannot distinguish between a skill used **yesterday** and a skill used **10 years ago**.
    - *Scenario:* A candidate was a "Java Developer" in 2010 but switched to "HR Management" in 2015. Their resume contains heavy keywords for both.
    - *Result:* The system sees a mixed vector. If the user is hiring for a Java role, this candidate might score highly despite not having coded in a decade.
- **Limitation:** The system measures **Cumulative Life Experience**, not **Current Operational Readiness**. It lacks the logic to apply "Time-Decay" weighting to older skills, which is a standard heuristic used by human recruiters.

Information Loss in Visual Signifiers
Candidates increasingly rely on non-textual visual cues to convey information, which the text extraction module is blind to.

- **The "Skill Bar" Blind Spot:**
  - *Scenario:* A candidate lists "Python" followed by a visual progress bar filled to 20% (Beginner) or 100% (Expert).
  - *System View:* The text extraction layer perceives only the string "Python." It cannot "see" the graphics.
- **Consequence:** The system scores the Beginner and the Expert identically regarding keyword presence. It creates a **Competence Flattening** effect where mere *exposure* to a tool is conflated with *mastery* of it. Similarly, icons representing "Email," "Phone," or "GitHub" are lost, sometimes causing the system to miss contact information if it relies purely on icon recognition.

The Algorithmic Gap: Prototype Dependencies and Bias

The core innovation of the proposed system—**Prototype-Based Modeling**—introduces its own set of unique algorithmic constraints. Unlike a human recruiter who can adapt to new information dynamically, the system is a rigid comparator. It measures the distance between a candidate and a static "Master Profile." Therefore, the system is only as intelligent, flexible, and fair as the artifacts generated during its training phase.
The "Cold Start" and Out-of-Distribution Problem
The system relies on the existence of a pre-calculated centroid (prototype) for every valid job category. This creates a significant dependency on the completeness of the training data.
1. The Novel Role Limitation The system lacks the capability for **Zero-Shot Learning**. It cannot infer selection criteria from a job description alone; it requires a history of resumes to build a prototype.

- **The Scenario:** If an organization wishes to hire for a completely novel or emerging role—for example, a "Prompt Engineer," "Quantum Algorithmist," or "Sustainable Energy Consultant"—and the training dataset (*AzharAli05*) contains zero examples of this category, the system **cannot** function for that role.
- **The "Force-Fit" Error:** Instead of flagging the resume as "Unknown," the K-Nearest Neighbors (KNN) classifier is mathematically compelled to classify the applicant into the nearest *existing* cluster.
  - *Failure Mode:* A "Quantum Physicist" might be misclassified as a "Data Scientist" simply because they share math-related stop-words.

This leads to an invalid suitability score, as the candidate is being measured against the wrong yardstick.

2. Static Nature of Prototypes Because the prototypes are generated offline and serialized (prototypes.pkl), they do not update automatically.

- **The "Frozen Time" Constraint:** As the industry evolves, the definition of a role changes. A "Frontend Developer" prototype trained on data from 2018 might emphasize "jQuery" and "Bootstrap." A candidate in 2025 using "Next.js" and "Tailwind" might score lower because their vocabulary diverges from the older prototype.
- **Maintenance Overhead:** To fix this, the system requires manual retraining cycles. It does not possess an **Online Learning** feedback loop to self-correct based on recruiter actions in real-time.

The Synonym Penalty (Vocabulary Rigidity)

The use of **Keyword Intersection** as a dedicated feature in the scoring algorithm enhances interpretability but enforces a penalty on vocabulary diversity.

1. The Mechanism of Penalty The "Master Profile" is built from the aggregate vocabulary of the training data. The system effectively standardizes the jargon used by the majority.

- **Scenario:** If 80% of "Web Developers" in the training data use the term **"ReactJS,"** the prototype vector encodes "ReactJS" as the standard.
- **The Deviation:** A highly qualified candidate who uses the synonymous term **"React.js"** or simply **"React"**—and whose text cleaning did not perfectly normalize this—will suffer a reduction in their Keyword Match score.
- **Limitation:** Unlike the Semantic Vector (Cosine Similarity), which can handle some degree of relatedness, the explicit Keyword Intersection feature is binary and rigid. This bias forces candidates to conform to the specific jargon of the training set to achieve maximum scores, potentially penalizing linguistic creativity or alternative nomenclatures (e.g., "UI Engineering" vs. "Frontend Development").

Historical Bias Propagation (The Echo Chamber)

The "Master Prototypes" are not objective standards of excellence; they are statistical averages of historical hiring data. Therefore, the system is subject to **Algorithmic Bias**.

1. Bias Encoding If the historical training data for "Engineering Manager" is heavily biased toward a specific demographic (e.g., 90% male), the resume text may contain subtle gendered linguistic markers.

- **Linguistic Markers:** Research shows that male-dominated corpora often feature more aggressive action verbs (e.g., "crushed," "dominated,"

"tackled"), while female-authored resumes may prioritize collaborative terms (e.g., "supported," "facilitated," "nurtured").
- **The "Ideal" Candidate:** The generated prototype will encode the majority markers as part of the "Ideal Candidate" vector.

2. Bias Amplification

Consequently, the system may unintentionally penalize female applicants or those from different cultural backgrounds. This occurs not because they lack technical skills, but because their narrative style diverges from the historically biased centroid.

- **The Recursive Trap:** The system does not possess an external, objective definition of "Fairness" or "Quality." Its definition of "Good" is recursive: *"Good is what looked like the resumes we hired in the past."* This creates an **Echo Chamber** that reinforces the status quo rather than promoting diversity or discovering non-traditional talent.

The Veracity Gap: Truthfulness and Adversarial Vulnerability

Finally, the system operates under a fundamental assumption of **Data Veracity**. It evaluates the *content* of the resume but possesses no mechanism to verify the *truth* of that content. This exposes the system to security vulnerabilities and manipulation.

Vulnerability to Keyword Stuffing Attacks

Since the system is known to reward high keyword overlap and semantic density, it is highly susceptible to **Adversarial Examples** and "Black Hat" optimization techniques.

1. The "Invisible Text" Attack

- **The Attack Vector:** A candidate can copy the entire job description or a list of high-value keywords (e.g., "Python, AWS, Docker, Kubernetes, AI, ML") and paste them into the footer of their PDF. They then set the font size to 1pt and the color to white.
- **The System Failure:** To a human recruiter viewing the PDF, this text is invisible. However, the PyPDF2 extraction module reads the raw character stream, where color and font size are usually ignored.
- **Result:** The TF-IDF vectorizer counts these terms as valid input. The Gradient Boosting Regressor, seeing a perfect keyword match, awards a near-perfect suitability score (e.g., 99%).
- **Constraint:** Without a "Semantic Coherence" checker or a computer-vision-based layout analyzer to detect hidden text, the system can be easily gamed by savvy applicants, rendering the score meaningless.

The "Hallucination of Competence"

The system measures the *presence* of claims, not the *validity* of skills.

- **The Unverified Claim:** A candidate who writes *"I architected the entire Google Search backend"* contributes the same semantic weight to the vector as a candidate who actually did so.
- **Lack of External Verification:** The system has no API connections to external truth sources (e.g., LinkedIn API, University Registrars, Criminal Background Checks). It cannot verify employment dates, degree authenticity, or skill proficiency.
- **Operational Risk:** This limits the system to assessing **Potential Suitability based on Self-Reported Claims**. It effectively measures how well a candidate can *write* a resume, which is not always perfectly correlated with how well they can *perform* the job.

In conclusion, while the developed AI-driven Applicant Tracking System represents a significant advancement in automated, interpretable screening, it operates within a bounded reality. It is an **English-language, text-based decision support tool** that excels at pattern matching but lacks cognitive understanding, temporal awareness, and truth verification.
The system is best understood as a **High-Speed Filter**—capable of sorting through thousands of applicants to identify probable matches—rather than a **Final Decision Maker**. It successfully automates the "rote" aspect of screening but requires human oversight to mitigate its semantic blind spots, check for adversarial manipulation, and ensure that historical biases are not perpetuated in future hiring decisions. These limitations define the necessary "Human-in-the-Loop" protocol required for its responsible deployment.

Conclusion

Summary of Achievement

This research successfully designed, developed, and validated an intelligent, AI-driven Applicant Tracking System (ATS) aiming to resolve the operational bottlenecks of high-volume digital recruitment. Moving beyond the rigid, keyword-based filtering of traditional systems, this project implemented a Hybrid Classification-Regression Architecture that utilizes Natural Language Processing (NLP) to evaluate candidates based on semantic alignment rather than simple string matching.

By integrating K-Nearest Neighbors (KNN) for domain identification and Gradient Boosting for suitability scoring, the system effectively automates the extraction, normalization, and evaluation of unstructured resume data. A core innovation of this work was the introduction of Prototype-Based Modeling, where "Master Profiles" generated from historical data serve as dynamic benchmarks, allowing the system to adapt to evolving industry standards without manual rule updates.

Key Findings and Performance

The experimental results and system validation highlight several critical outcomes:

- **High Classification Accuracy:** The system demonstrated a classification accuracy exceeding 98% on the test dataset, validating the efficacy of the One-vs-Rest KNN strategy for differentiating between 45 distinct technical roles.
- **Interpretable Scoring:** The implementation of "Glass-Box" metrics—specifically separating Content Match (Cosine Similarity) from Keyword Overlap—successfully addressed the "Black Box" problem common in AI, providing recruiters with transparent rationales for every score.
- **Operational Efficiency:** The stateless, prototype-driven architecture proved capable of processing resumes in real-

time using standard computing hardware, confirming the feasibility of the system as a lightweight, deployable decision support tool.

Operational Boundaries and Limitations

While the system represents a significant advancement over manual screening, it operates within strictly defined boundaries to ensure reliability:

- **Domain Specificity:** The system is optimized exclusively for the Information Technology sector, relying on the high-signal vocabulary of technical roles. It is not designed to evaluate non-technical domains (e.g., Creative Writing, Medicine) where vocabulary is less standardized.
- **The Semantic Gap:** As a statistical pattern matcher, the system lacks cognitive understanding. It is subject to "Bag-of-Words blindness," where it may fail to distinguish between a user managing a tool versus being managed by it, or misinterpret negated sentences.
- **Veracity Gap:** The system evaluates the presence of skills but cannot verify the truth of claims, remaining vulnerable to adversarial attacks like "invisible text" keyword stuffing.

Future Scope

To overcome these limitations and further enhance the system's capabilities, future work should focus on:

1. **Deep Learning Integration:** Replacing TF-IDF with Transformer-based embeddings (e.g., BERT or RoBERTa) to capture contextual nuance and resolve the semantic gap.
2. **Multilingual Support:** Expanding the preprocessing pipeline to support non-English languages and globalize the system's applicability.
3. **Layout Analysis:** Integrating Computer Vision (OCR) to preserve the visual structure of resumes, preventing data corruption in complex multi-column layouts.

Final Concluding Remark

Ultimately, this project redefines the role of the ATS from a simple "gatekeeper" to an intelligent Decision Support System. By successfully automating the rote task of pattern matching while maintaining transparency, the system empowers human recruiters to focus on what they do best: assessing cultural fit, soft skills, and potential, thereby making the recruitment process faster, fairer, and more efficient.

References

1. Datasets

4. **Resume Screening Dataset.** (n.d.). Retrieved from GitHub (AzharAli05/Resume-Screening-Dataset).
    a. *Link:* https://github.com/AzharAli05/Resume-Screening-Dataset
    b. *Description:* A structured dataset containing resume text data labeled into 25 distinct job categories.

2. Key Algorithms & Theoretical Foundations

- **TF-IDF:** Salton, G., & Buckley, C. (1988). *Term-weighting approaches in automatic text retrieval*. Information Processing & Management.
    - *Link:* https://dl.acm.org/doi/10.1016/0306-4573(88)90021-0
- **K-Nearest Neighbors (KNN):** Cover, T., & Hart, P. (1967). *Nearest neighbor pattern classification*. IEEE Transactions on Information Theory.
    - *Link:* https://ieeexplore.ieee.org/document/1053964
- **Gradient Boosting:** Friedman, J. H. (2001). *Greedy function approximation: A gradient boosting machine*. Annals of Statistics.
    - *Link:* [suspicious link removed]
- **One-vs-Rest Strategy:** Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
    - *Link:* https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf

3. Software Libraries & Frameworks

1. **Scikit-learn:** Pedregosa, F., et al. (2011). *Scikit-learn: Machine Learning in Python*.
    a. *Link:* https://scikit-learn.org/stable/index.html
2. **Pandas:** McKinney, W. (2010). *Data Structures for Statistical Computing in Python*.
    a. *Link:* https://pandas.pydata.org/docs/
3. **NLTK (Natural Language Toolkit):** Bird, S., Klein, E., & Loper, E. (2009).
    a. *Link:* https://www.nltk.org/

Bibliography

- **Scikit-learn Official Documentation.** *Supervised learning: Nearest Neighbors and Multiclass algorithms.*
  - *Link:* https://scikit-learn.org/stable/modules/neighbors.html
- **Python Software Foundation.** *Python 3.10 Documentation.*
  - *Link:* https://docs.python.org/3/
- **GeeksforGeeks.** *Resume Screening with Python.* (Referenced for preprocessing pipelines).
  - *Link:* https://www.geeksforgeeks.org/resume-screening-with-python/
- **Towards Data Science.** *Understanding TF-IDF and Cosine Similarity for Text Comparison.*
  - *Link:* https://towardsdatascience.com/understanding-tf-idf-and-cosine-similarity-d04b901584

Appendix

Key Source Code Modules:

Appendix A: Key Source Code Modules

A.1 Text Cleaning Logic *From app.py. This function standardizes input text by removing URLs and special characters.*

```
def clean_text(txt):
    txt = re.sub(r'http\S+\s', ' ', txt)
    txt = re.sub(r'[^\w\s]', ' ', txt)
    return txt.lower()
```

A.2 Prototype Generation *From train_model.py. This logic aggregates resumes by category to create the "Master Profile."*

```
prototypes = df.groupby(label_col)['cleaned_resume'].apply(lambda x: ' '.join(x)).to_dict()
pickle.dump(prototypes, open('prototypes.pkl', 'wb'))
```

A.3 The Scoring Engine *From app.py. This function calculates the final scores using Cosine Similarity, Keyword Intersection, and the AI model fallback logic.*

```
def calculate_scores(text, category):
    master_profile = prototypes[category]
    cleaned_resume = clean_text(text)

    vecs = tfidf.transform([cleaned_resume, master_profile])
    cosine_sim = cosine_similarity(vecs[0], vecs[1])[0][0]

    res_tokens = set(cleaned_resume.split())
    mp_tokens = set(master_profile.split())
    keyword_match = len(res_tokens.intersection(mp_tokens)) / len(mp_tokens) if mp_tokens
else 0
```

```
    ml_score = ats_model.predict([[cosine_sim, keyword_match]])[0]

    if ml_score < 10:
        final_score = cosine_sim * 100
    else:
        final_score = ml_score

    return round(final_score, 1), round(cosine_sim*100, 1), round(keyword_match*100, 1)
```

A.4 Model Training Configuration *From train_model.py. The specific parameters used for Vectorization and the K-Nearest Neighbors classifier.*

```
tfidf = TfidfVectorizer(stop_words='english', max_features=5000)
clf = OneVsRestClassifier(KNeighborsClassifier())
clf.fit(requiredText, df['Category_ID'])
```

Here is the continuation of your Appendix with the remaining critical code modules (Gradient Boosting logic, PDF parsing, and the Main Flask Route).

A.5 Gradient Boosting Regressor Training

From train_model.py. This snippet demonstrates how the "Judge" (ATS Scoring Model) is trained using synthetic data to learn the relationship between similarity metrics and final scores.

Python

```
# Synthetic training data for the Regressor (Simulating human grading)
# Features: [Cosine Similarity, Keyword Match]
X_reg = np.array([
    [0.9, 0.8], [0.8, 0.7], [0.7, 0.6], [0.6, 0.5],
    [0.5, 0.4], [0.4, 0.3], [0.3, 0.2]
])
# Target: [ATS Score (0-100)]
y_reg = np.array([95, 85, 75, 65, 55, 45, 35])

ats_model = GradientBoostingRegressor()
ats_model.fit(X_reg, y_reg)
```

```
pickle.dump(ats_model, open('ats_score_model.pkl', 'wb'))
```

A.6 PDF Extraction Logic

From app.py. The utility function that handles file uploads, specifically extracting raw text from PDF documents using PyPDF2.

Python

```
def input_pdf_text(uploaded_file):
    reader = pdf.PdfReader(uploaded_file)
    text = ""
    for page in range(len(reader.pages)):
        page = reader.pages[page]
        text += str(page.extract_text())
    return text
```

A.7 Main Prediction Route

From app.py. The central controller that orchestrates the entire workflow: receiving the file, cleaning it, classifying the role (KNN), and calculating the score (Gradient Boosting).

Python

```
@app.route('/pred', methods=['POST'])
def pred():
    if 'resume' in request.files:
        file = request.files['resume']
        text = input_pdf_text(file)  # Extract
        cleaned_text = clean_text(text)  # Clean

        # 1. Transform & Classify (KNN + OvR)
        input_features = tfidf.transform([cleaned_text])
        prediction_id = clf.predict(input_features)[0]
        category_name = category_mapping[prediction_id]

        # 2. Score (Master Profile + Gradient Boosting)
        final_score, cosine, keyword = calculate_scores(cleaned_text, category_name)

        return render_template('result.html',
                               prediction=category_name,
                               score=final_score)
```

Project Structure

A high-level view of the directory organization.

Plaintext

```
Resume-Screening-System/
├── app.py                # Main Flask Application (Runtime)
├── train_model.py        # Training Script (Offline Phase 1)
├── models/
│   ├── model.pkl         # Trained KNN Classifier
│   ├── tfidf.pkl         # Fitted TF-IDF Vectorizer
│   ├── prototypes.pkl    # Master Profiles Dictionary
│   └── ats_score_model.pkl  # Gradient Boosting Regressor
├── templates/
│   ├── index.html        # Upload Page
│   └── result.html       # Dashboard Page
├── static/               # CSS and Styles
└── requirements.txt      # Dependencies (Flask, scikit-learn, PyPDF2)
```