International Centre for Education and Research (ICER)
VIT-Bangalore

Analyzing Technological Trends in Autonomous Vehicles through Patent Text Mining and Predictive Modeling

CS7610 – CAPSTONE PROJECT

REPORT

Submitted by

Aniket Shinde – 24MSP3038

In partial fulfilment for the award of the degree of

POST GRADUATE PROGRAMME

INTERNATIONAL CENTRE FOR HIGHER EDUCATION AND RESEARCH

VIT BANGALORE

June, 2025

International Centre for Education and Research (ICER)
VIT-Bangalore

BONAFIDE CERTIFICATE

Certified that this project report **“Analyzing Technological Trends in Autonomous Vehicles through Patent Text Mining and Predictive Modeling**

” is the bonafide record of work done by **“Aniket Shinde – 24MSP3038”** who carried out the project work under my supervision.

Signature of the Supervisor	**Signature of Director**
Prof. Ramya Mohanakrishnan	**Prof. Prema M**
Assistant Professor,	**Director,**
ICER	ICER
VIT Bangalore	VIT Bangalore.

Evaluation Date:

International Centre for Education and Research (ICER)
VIT-Bangalore

ACKNOWLEDGEMENT

I express my sincere gratitude to our director of ICER **Prof. Prema M.** for their support and for providing the required facilities for carrying out this study.

I wish to thank my faculty supervisor(s), **Prof. Ramya Mohanakrishnan, Assistant Professor**, ICER for extending help and encouragement throughout the project. Without his/her continuous guidance and persistent help, this project would not have been a success for me.

I am grateful to all the members of ICER, my beloved parents, and friends for extending the support, who helped us to overcome obstacles in the study.

TABLE OF CONTENTS

ABSTRACT

This project presents a comprehensive analysis of the autonomous vehicle (AV) industry through patent intelligence, leveraging a novel unstructured-to-structured data pipeline that transforms raw patent text into actionable insights. Using 667 carefully filtered US AV patents from leading companies, I constructed an end-to-end NLP-to-ML framework encompassing semantic clustering, innovation trend analysis, technology emergence detection, and innovation outlier identification. The pipeline integrates advanced natural language processing techniques with machine learning models to reveal critical industry patterns including emerging technologies, declining innovations, company positioning, and strategic pivots. Through semantic clustering and temporal analysis, combined with LLM-enhanced data structuring, this work demonstrates how patent data can be systematically transformed to provide strategic intelligence for understanding technological evolution and competitive dynamics in the rapidly advancing autonomous vehicle sector.

CHAPTER 1

1. INTRODUCTION

1.1 Introduction to the Research Topic

The autonomous vehicle (AV) industry is one of the fastest-evolving sectors in technology, where cutting-edge innovations in AI, sensing, computation, and connectivity converge. With a high volume of patents being filed by global tech leaders, patents serve as a rich resource for tracing technological progress, identifying strategic priorities, and forecasting future trends.

1.2 Background and Rationale

Understanding innovation dynamics in AVs is challenging due to the volume and complexity of patent data. Conventional analytics often fall short in revealing semantic relationships, emerging technologies, or innovation leadership. This research bridges that gap using advanced NLP and machine learning techniques to semantically cluster patents, analyze innovation trends, detect outliers, and extract structured knowledge with the help of LLMs. The motivation is to develop an AI-powered framework that can generate actionable insights for researchers, strategists, and policymakers tracking AV innovation.

1.3 Research Questions

What are the major technological clusters present in the AV patent landscape?

Which concepts and terms are emerging, declining, or persistently dominant over time?

Can structured insights like novelty and problem statements be extracted at scale using LLMs?

Who are the innovation leaders, and what strategic shifts or outlier technologies are they pursuing?

1.4 Scope and Limitations

The study focuses on a filtered subset of 667 US patents filed by top companies in the AV domain from 2014 to 2022. The scope includes semantic clustering, trend analysis, LLM-assisted structuring, and outlier detection. Limitations include potential API constraints, imperfect NLP interpretations, and challenges in fully resolving noise in unstructured data.

2 LITERATURE REVIEW

Trappey et al. (2020) established foundational work in intelligent patent summarization using ML and NLP techniques, demonstrating automated transformation of unstructured patent text into structured insights. ***Cho et al. (2021)*** specifically examined autonomous driving patents through citation analysis, validating strategic importance of AV patent intelligence. ***Kim & Bae (2017)*** introduced forecasting methodologies for identifying promising technologies through patent analysis, aligning with my trend analysis components. ***Lattimer et al. (2023)*** contributed advanced NLP techniques for processing long documents, relevant to my full-text patent analysis pipeline. My work extends these foundations by integrating semantic clustering, LLM-enhanced structuring, and emergence detection into a comprehensive NLP-to-ML framework, moving beyond citation-based analysis to capture deeper technological relationships and innovation patterns in the AV domain.

3 OBJECTIVE

The primary objective of this research is to develop an end-to-end NLP-to-ML pipeline that can transform large-scale, unstructured patent text from the autonomous vehicle (AV) domain into structured, actionable intelligence. This system is intended to support deeper strategic analysis and decision-making by uncovering hidden patterns, emerging technologies, and innovation leaders in the AV space.

To achieve this, several specific objectives were pursued:

- **Data Infrastructure Development:** Assemble a curated dataset of 667 US patents relevant to AV technologies. This involved full-text retrieval via the Lens.org API, extensive metadata preprocessing, and structured JSONL formatting to support advanced natural language processing workflows.

- **Semantic Pattern Discovery:** Use NLP models like Sentence-BERT and clustering techniques (e.g., UMAP + HDBSCAN) to identify coherent semantic groupings of patents. These clusters reveal hidden relationships, thematic overlaps, and corporate focus areas.
- **Temporal Innovation Analysis:** Track technological evolution over time through curated term extraction and trend analysis. This allowed detection of both emerging and declining concepts using domain-specific insights.
- **LLM-Enhanced Structuring:** Employ the Gemini API to derive structured features such as novelty, problems addressed, and proposed solutions from free-text patent content.
- **Strategic Intelligence Extraction:** Apply machine learning methods including Prophet, DBSCAN, and anomaly detection to uncover innovation outliers, identify emerging technologies, and analyze shifts in competitive positioning.

4 PROPOSED METHODOLOGY

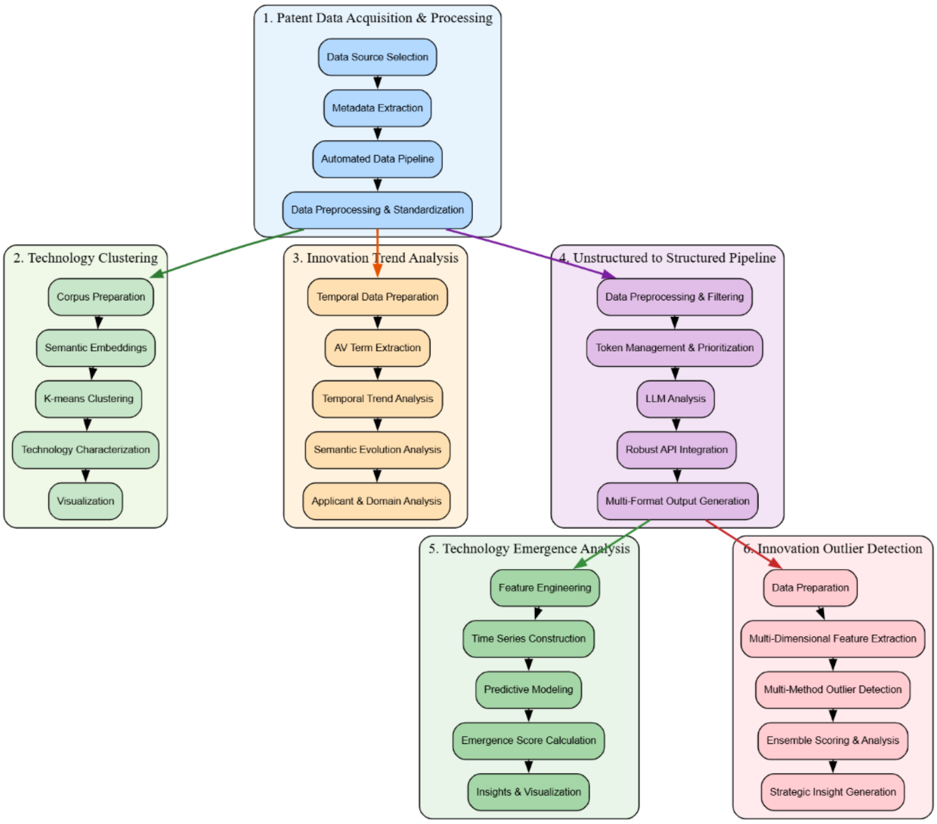


Fig 4.1 Methodology

Data Acquisition and Processing

Patent data was sourced from Lens.org, targeting 667 US patents from major AV industry players (Baidu, Ford, Amazon, Toyota, Google, etc.) published between 2020-2025. Strategic filters focused on machine learning and neural network technologies (CPC codes G06N3/08, G06N20/00) while excluding computer vision classifications. Complete patent records were retrieved via Lens.org REST API and stored in JSONL format. Raw data underwent preprocessing to remove unnecessary attributes, flatten nested structures, and standardize nine core fields: lens_id, publication date, claims, description, applicant name, CPC symbols, title, and abstract.

Technology Clustering Analysis

Patent texts (titles, abstracts, claims) were preprocessed and truncated to 500 words for semantic analysis. The Sentence-BERT model (all-MiniLM-L6-v2) generated 384-dimensional embeddings to capture contextual meaning beyond keyword matching. Optimal cluster numbers were determined using silhouette scores, Calinski-Harabasz indices, and elbow method analysis. K-means clustering grouped patents by semantic similarity, with clusters characterized through TF-IDF keyword extraction and validated via coherence analysis.

Innovation Trend Analysis

Meaningful technical terms were extracted using multi-layered NLP approaches: spaCy for noun phrase identification, regex patterns for AV-specific terminology, and n-gram analysis for compound terms. Generic patent language was filtered using comprehensive stopword lists. Temporal trends were analyzed through term frequency normalization, Pearson correlation analysis, and semantic evolution using SBERT embeddings to identify technological shifts over time.

Structured Data Transformation

Unstructured patent text was transformed into structured data using Gemini 1.5 Flash API with intelligent token management (3,000 tokens per patent). Each patent was analyzed for core innovation, conceptual categories, and AV technology area classification. The pipeline implemented robust error handling, progressive checkpointing every 50 patents, and multi-format output generation for downstream ML applications.

Emergence and Outlier Detection

Technology emergence scores integrated growth rates, applicant diversity, market concentration, and innovation quality metrics using Prophet and ensemble ML models. Outlier detection employed four complementary methods (Isolation Forest, Autoencoders, DBSCAN, statistical methods) to identify radical innovations through ensemble scoring, with the top 15% classified as breakthrough patents.

5 TOOLS AND TECHNIQUES

1. **Sentence-BERT (SentenceTransformer)**: A state-of-the-art transformer-based model for generating semantic embeddings that capture contextual meaning in patent text. The 'all-MiniLM-L6-v2' model was employed to create 384-dimensional vector representations enabling sophisticated similarity analysis beyond traditional keyword matching.
2. **Scikit-learn**: A comprehensive machine learning library providing clustering algorithms (K-means, DBSCAN, Agglomerative Clustering), dimensionality reduction techniques (PCA, t-SNE), and evaluation metrics (silhouette scores, Calinski-Harabasz indices). Additional modules included TF-IDF vectorization for text analysis and ensemble methods for outlier detection.
3. **UMAP (Uniform Manifold Approximation and Projection)**: An advanced dimensionality reduction technique used for creating interpretable 2D visualizations of high-dimensional semantic embeddings, preserving both local and global structure in the patent technology landscape.
4. **spaCy**: An industrial-strength NLP library employed for sophisticated text preprocessing, named entity recognition, part-of-speech tagging, and compound term extraction from patent documents.

5. **TensorFlow/Keras**: Deep learning frameworks utilized for building autoencoder neural networks for anomaly detection and LSTM models for time series forecasting in technology emergence analysis.
6. **Google Gemini API**: Large language model integration for transforming unstructured patent text into structured data through intelligent content analysis and categorization.
7. **NLTK (Natural Language Toolkit)**: Comprehensive NLP library providing tokenization, stemming, lemmatization, and stopword filtering capabilities for text preprocessing and meaningful term extraction.
8. **Prophet (Facebook's Time Series Library)**: Specialized forecasting tool for decomposing patent filing trends, detecting seasonality patterns, and predicting future technology emergence trajectories.
9. **Plotly**: Interactive visualization library enabling creation of dynamic, multi-dimensional plots for exploring patent landscapes, technology clusters, and temporal trends with enhanced user interactivity.
10. **Isolation Forest**: An unsupervised anomaly detection algorithm specifically designed for identifying outliers in high-dimensional spaces, crucial for detecting radical innovation patents.
11. **Ensemble Methods**: Integration of Random Forest and Gradient Boosting regressors for robust predictive modeling, combining multiple weak learners to improve forecasting accuracy and reduce overfitting.
12. **Statistical Analysis Tools**: SciPy's statistical functions for Pearson correlation analysis, hypothesis testing, and advanced statistical validation of trend patterns and emergence scores.

6 IMPLEMENTATION

6.1 DATA ACQUISITION AND PROCESSING PIPELINE

6.1.1 PATENT DATA SOURCE AND COLLECTION

The patent data was sourced from Lens.org, a comprehensive global patent database providing access to over 130 million patent records worldwide. Strategic filtering was applied to focus specifically on autonomous vehicle technologies from major industry players. The filtering criteria included:

Publication Period: 2020-2025 to capture recent technological developments **Patent Status**: Only granted US patents with complete documentation

Legal Status: Active patents to ensure current relevance

Target Companies: Baidu USA LLC, Ford Global Tech LLC, Amazon Tech INC, Qualcomm INC, Toyota Motor CO LTD, Intel CORP, Google LLC, and Nvidia CORP

Classification Codes: CPC codes targeting machine learning (G06N3/08, G06N20/00, G06N3/045) while excluding certain computer vision classifications

This strategic filtering process yielded a curated dataset of 667 patents from hundreds of thousands of available records, ensuring high relevance for autonomous vehicle technology analysis.

6.1.2 API INTEGRATION AND DATA RETRIEVAL

Initial metadata export from Lens.org provided essential identifiers in CSV format (lens-export.csv), particularly the critical Lens IDs required for comprehensive data retrieval. However, this export lacked detailed patent information such as full claims and technical descriptions.

To obtain complete patent records, API access was requested and granted for Lens.org's REST API. A systematic automated pipeline was developed to:

- Fetch complete patent records using collected Lens IDs
- Handle API rate limiting and error responses
- Store raw data in JSONL format (av_patentdata.jsonl) for efficient processing
- Maintain data integrity across large-scale retrieval operations

6.1.3 DATA PREPROCESSING AND STANDARDIZATION

Raw API data contained extensive nested attributes and irrelevant metadata including patent families, legal agents, and administrative details. A comprehensive preprocessing pipeline was implemented to:

Data Cleaning: Remove unnecessary attributes and metadata not relevant to technical analysis

Structure Flattening: Convert nested JSON structures to accessible top-level keys
Attribute Standardization: Ensure consistent availability of core attributes across all patent records

Quality Validation: Verify completeness of essential fields for downstream analysis

The final standardized schema included nine core attributes essential for NLP and clustering analysis:

- lens_id (unique identifier)

- date_published (temporal analysis)
- claims (technical specifications)
- description (detailed technical content)
- earliest_claim_date (priority timeline)
- applicant_name (company attribution)
- cpc_symbols (classification codes)
- invention_title_text (innovation summary)
- abstract_text (technical overview)

6.2 SEMANTIC CLUSTERING ANALYSIS

6.2.1 CORPUS PREPARATION AND TEXT PREPROCESSING

Patent documents were prepared for semantic analysis by combining three key textual components: invention titles, abstracts, and claims. This multi-component approach ensures comprehensive capture of both high-level innovation concepts and detailed technical specifications.

Text preprocessing operations included:

- Cleaning technical references (e.g., "Figure 1" → "figure")
- Removing excessive whitespace and formatting artifacts
- Truncating documents to 500 words to optimize processing efficiency
- Filtering documents with insufficient content (<100 characters)
- Preserving technical terminology and domain-specific language

6.2.2 SEMANTIC EMBEDDING GENERATION

The Sentence-BERT (SBERT) model was employed to transform patent texts into high-dimensional semantic embeddings that capture contextual meaning beyond traditional keyword matching approaches.

Model Specification: 'all-MiniLM-L6-v2' architecture

Embedding Dimensions: 384-dimensional vectors with L2 normalization **Similarity Measurement**: Cosine similarity calculations for semantic relationships **Advantage**: Captures nuanced technological relationships that TF-IDF methods typically miss

This approach enables identification of patents with similar technological concepts even when using different terminology or technical language.

6.2.3 CLUSTERING PARAMETER OPTIMIZATION

Multiple clustering quality metrics were employed to determine optimal cluster configurations:

Silhouette Analysis: Measures intra-cluster cohesion versus inter-cluster separation

Calinski-Harabasz Index: Evaluates cluster variance ratios for optimal separation
Elbow Method: Identifies diminishing returns in inertia reduction

Visualization: Metric plots guide informed decision-making for cluster number selection

6.2.4 K-MEANS CLUSTERING IMPLEMENTATION

K-means clustering was applied to group patents based on semantic similarity in the embedding space. The implementation included:

- Cluster quality validation through silhouette scoring
- Coherence analysis measuring average cosine similarity within clusters
- Alternative method validation using hierarchical clustering and DBSCAN
- TF-IDF-based keyword extraction for cluster characterization

6.3 INNOVATION TREND ANALYSIS

6.3.1 TECHNICAL TERM EXTRACTION

A multi-layered approach was implemented to extract technically relevant terms from patent documents:

NLP-based Extraction:

- spaCy library for noun phrase identification
- Compound term detection preserving grammatical structures
- Part-of-speech tagging for technical relevance validation

Domain-specific Pattern Matching:

- Regex patterns targeting AV-specific terminology
- Coverage across perception systems, localization, path planning
- V2X communication and safety standards terminology

Technical N-gram Analysis:

- 2-3 word combinations with technical relevance

- POS tagging pattern validation
- Compound technical term preservation

6.3.2 TERM FILTERING AND QUALITY CONTROL

Comprehensive filtering mechanisms ensure extraction of meaningful technical terms:

- Generic patent language removal using extensive stopword lists
- Boilerplate term filtering specific to patent documentation
- Technical keyword relevance scoring
- Proper phrase structure validation
- Awkward phrase detection and elimination

6.3.3 TEMPORAL TREND ANALYSIS

Time-based analysis segments were created to track innovation evolution:

- Yearly/quarterly segmentation with minimum patent thresholds
- Term frequency normalization by patent count for fair comparison
- Pearson correlation analysis for trending term identification
- Statistical significance testing for trend validation
- Semantic evolution analysis using SBERT embeddings

6.4 UNSTRUCTURED TO STRUCTURED DATA TRANSFORMATION

6.4.1 LLM-BASED ANALYSIS PIPELINE

A sophisticated pipeline was developed to transform unstructured patent text into structured analytical data using Google's Gemini 1.5 Flash API.

Content Prioritization System:

- Token limit management (3,000 tokens per patent)
- Hierarchical content inclusion: title (always) → abstract → claims (truncated if necessary)
- Intelligent truncation preserving critical technical information

6.4.2 STRUCTURED EXTRACTION FRAMEWORK

Each patent underwent comprehensive analysis extracting four key analytical components:

Core Innovation Analysis:

- Problem identification and technical challenges addressed
- Proposed solution methodology and approach
- Novelty aspects and technological advancement
- Technical implementation details

Conceptual Categorization:

- Primary technology classification with confidence scoring
- Secondary category identification for cross-domain innovations
- Technology convergence pattern recognition

AV Technology Domain Classification:

- Systematic classification into predefined autonomous vehicle domains
- Coverage of perception, localization, control systems, communication
- Multi-domain patent identification for convergence analysis

6.4.3 ROBUST API INTEGRATION

The implementation included comprehensive error handling and reliability mechanisms:

- Exponential backoff for server errors and temporary failures
- Rate limit respect and automatic retry mechanisms
- Conservative request delays preventing API exhaustion
- Progressive checkpointing every 50 patents for resumption capability
- Multi-format output generation (CSV, Excel, JSON, Pickle)

6.5 TECHNOLOGY EMERGENCE AND OUTLIER DETECTION

6.5.1 EMERGENCE SCORING METHODOLOGY

A composite scoring system was developed to identify emerging technologies within the autonomous vehicle patent landscape:

Growth Rate Analysis: Recent versus historical patent activity comparison **Acceleration Metrics**: Second derivative calculations of patent trend trajectories **Market Diversity**: Applicant diversity and competitive landscape analysis **Innovation Quality**: Average technical complexity and token density measurements

Classification Confidence: Technology categorization certainty scores

6.5.2 PREDICTIVE MODELING FRAMEWORK

Two complementary forecasting approaches were implemented:

Prophet Time Series Analysis:

- Automatic seasonality detection and trend decomposition
- Holiday and irregular event impact modeling
- Uncertainty interval estimation for predictions

Ensemble Machine Learning:

- Random Forest and Gradient Boosting regressors
- Feature engineering including lagged variables and rolling statistics
- Cross-validation for model robustness across different time periods

6.5.3 MULTI-METHOD OUTLIER DETECTION

Four complementary algorithms were employed for comprehensive outlier identification:

Isolation Forest: General anomaly detection through random partitioning **Autoencoder Neural Networks**: Reconstruction-based outlier identification **DBSCAN Clustering**: Identification of patents in sparse density regions

Statistical Methods: Mahalanobis distance for multivariate outlier detection

Results from all methods were normalized and combined using weighted averaging to create robust outlier scores, with the top 15% of patents classified as radical innovations.

6.6 VISUALIZATION AND REPORTING FRAMEWORK

6.6.1 DIMENSIONALITY REDUCTION AND VISUALIZATION

Advanced visualization techniques were implemented to create interpretable representations:

- UMAP and t-SNE dimensionality reduction for 2D technology landscape visualization
- Interactive clustering visualizations showing patent distributions
- Temporal evolution animations for trend identification
- Multi-dimensional scatter plots for emergence-growth analysis

6.6.2 COMPREHENSIVE REPORTING SYSTEM

The implementation generates detailed analytical reports including:

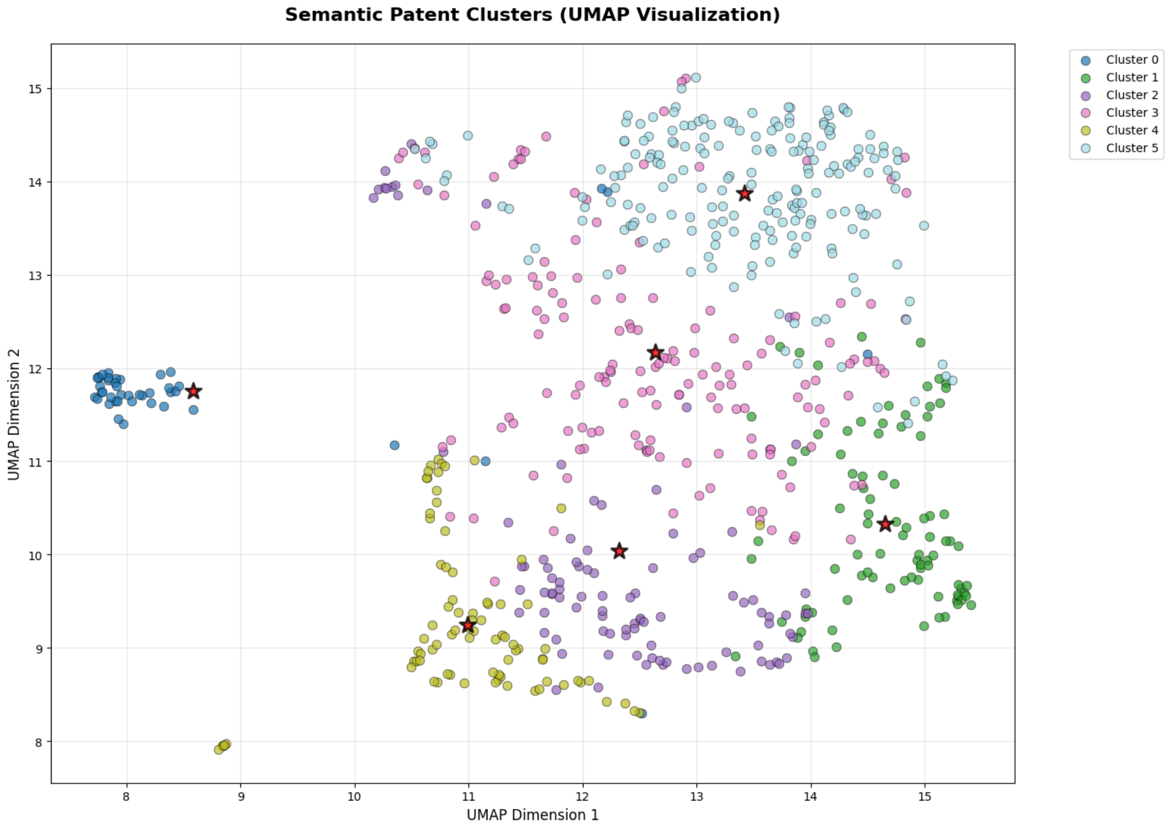
- Technology cluster characterization and market analysis
- Innovation trend timelines with statistical significance
- Competitive landscape analysis and market concentration metrics
- Emergence ranking with composite scoring explanations
- Strategic insights for technology investment decisions

7. RESULTS AND DISCUSSIONS

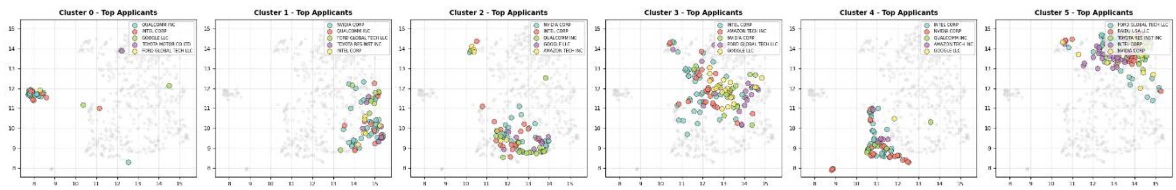
7.1 Technology Clustering (NLP)

Cluster ID	Technology Focus	Patents	Key Technologies	Semantic Coherence	Market Concentration
0	Wireless Communication	43	V2X, CSI, Wireless	0.47	High
1	Computer Vision	90	3D Vision, Depth, Camera	0.454	Medium
2	Neural Networks	99	Neural Arch, Deep Learning	0.435	Medium
3	AI/ML Applications	155	Data Models, Object Recognition	0.393	Low
4	Hardware Acceleration	83	Circuits, Processing, Memory	0.416	High
5	Vehicle Systems	197	Autonomous Driving, Sensors	0.47	Medium

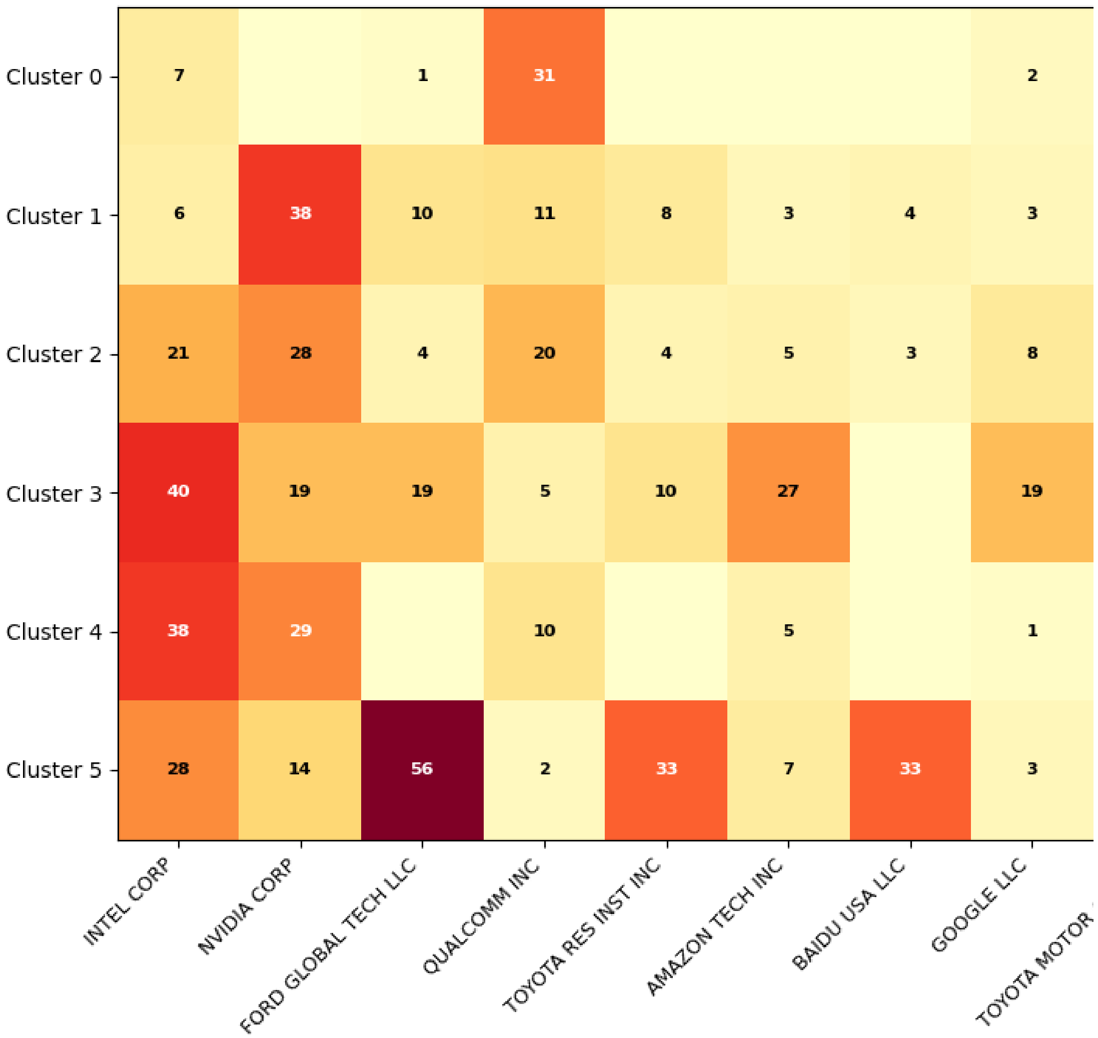
Technology Cluster	Market Leader	Leader Share (%)	Patents	Key Competitors
Wireless Communication	QUALCOMM INC	72.1	31	Intel (16.3%), Google (4.7%)
Computer Vision	NVIDIA CORP	42.2	38	Qualcomm (12.2%), Ford (11.1%)
Neural Networks	NVIDIA CORP	28.3	28	Intel (21.2%), Qualcomm (20.2%)
AI/ML Applications	INTEL CORP	25.8	40	Amazon (17.4%), Nvidia (12.3%)
Hardware Acceleration	INTEL CORP	45.8	38	Nvidia (34.9%), Qualcomm (12.0%)
Vehicle Systems	FORD GLOBAL TECH LLC	28.4	56	Baidu USA (16.8%), Toyota Res (16.8%)



The UMAP visualization displays distinct, well-separated clusters with minimal overlap, indicating good differentiation between semantic groups of patents, while the presence of some internal scatter within clusters suggests a degree of semantic breadth within each defined topic.



The multiple UMAP plots reveal that different top applicants (e.g., Intel, Nvidia, Ford) dominate distinct semantic spaces, suggesting specialized innovation focuses across the identified patent clusters.



The heatmap reveals that while Intel and Nvidia are broadly present across multiple clusters, Ford Global Tech LLC exhibits a particularly strong concentration in Cluster 5, and Toyota Research Institute Inc. shows a notable presence in Cluster 0 and 5, suggesting distinct areas of specialized innovation for different applicants.

Metric	Value
Total Patents Analyzed	667
Technology Clusters Identified	6
Unique Applicant Companies	14
Innovation Timeline	2014-2022 (8+ years)
Most Active Cluster	Vehicle Systems (197 patents)
Highest Market Concentration	72.1% (Qualcomm in Wireless)

• 6 distinct AV technology clusters identified across 667 patents

• Vehicle Systems largest cluster (197 patents, 29.5% of total)

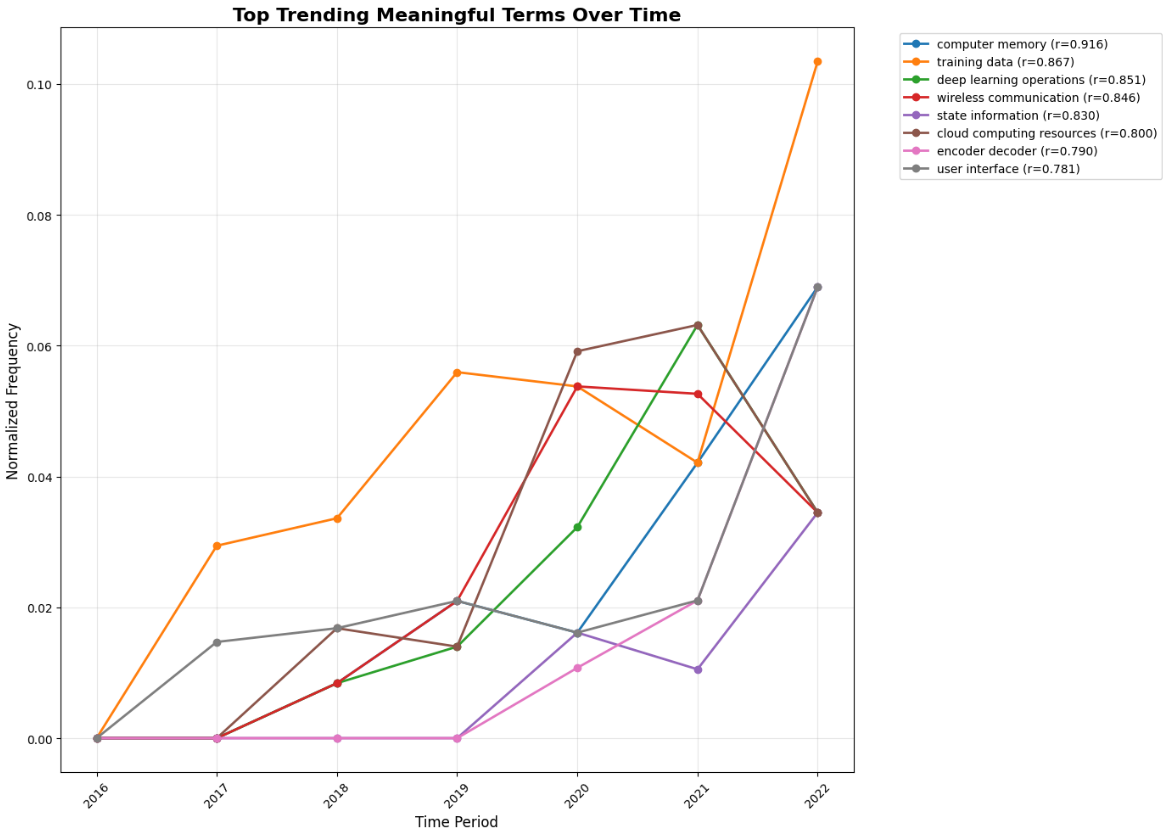
• Intel most diversified player (active in all 6 clusters)

• Wireless & Hardware show highest market concentration

• 8+ year innovation timeline (2014-2022) shows sustained R&D

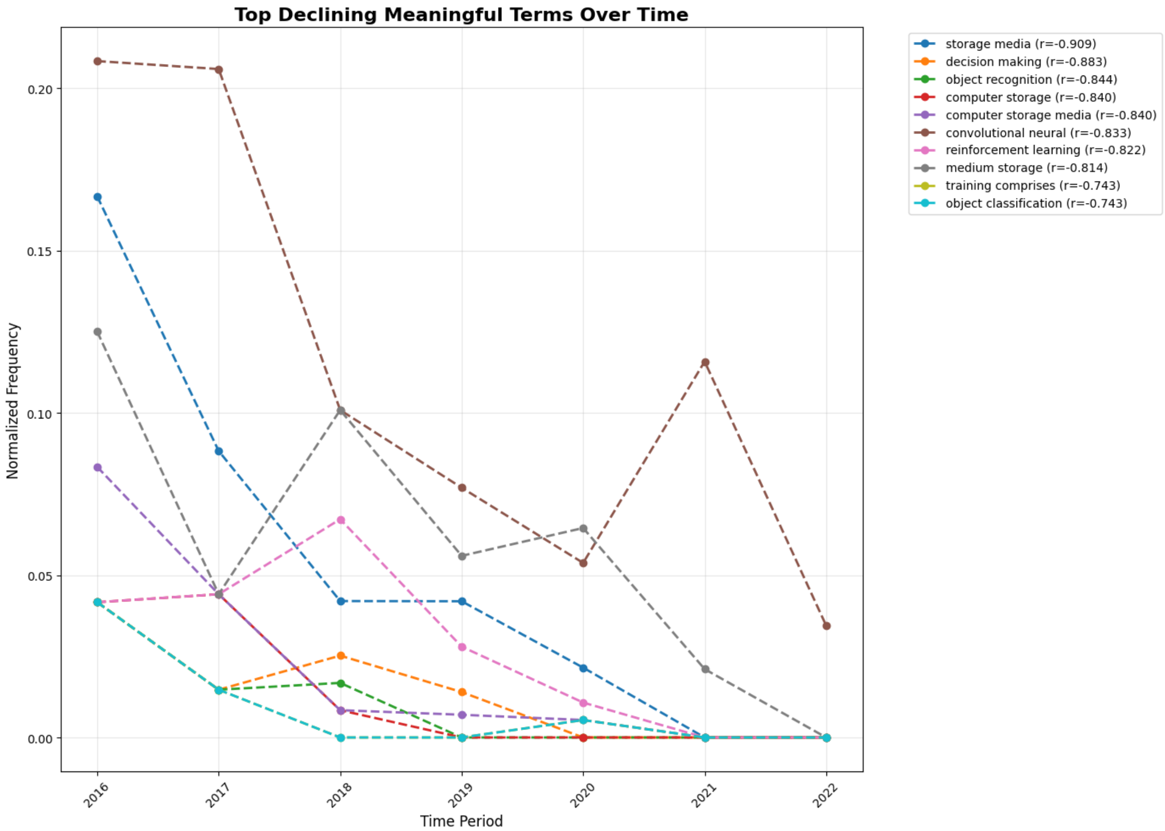
• Technology areas range from low-level hardware to high-level AI applications

7.2 Innovation Trend Analysis (NLP)



Top Emerging Technology Trends

Technology Term	Correlation (r)	Significance (p)	Trend
Computer Memory	0.916	0.004	↗ Strong Growth
Training Data	0.867	0.012	↗ Strong Growth
Deep Learning Operations	0.851	0.015	↗ Strong Growth
Wireless Communication	0.846	0.016	↗ Strong Growth
Cloud Computing Resources	0.800	0.031	↗ Growing



Top Declining Technology Trends

Technology Term	Correlation (r)	Significance (p)	Trend
Storage Media	-0.909	0.005	↘ Strong Decline
Decision Making	-0.883	0.008	↘ Strong Decline
Object Recognition	-0.844	0.017	↘ Strong Decline
Convolutional Neural	-0.833	0.020	↘ Declining
Reinforcement Learning	-0.822	0.023	↘ Declining

Leading Applicants & Focus Areas

Company	Patents	Primary Focus Areas
Intel Corp	140	Machine Learning, Collaborative Semantic Mapping, Memory Map
NVIDIA Corp	128	Encryption Standard, Operation Descriptor, Circuit ASIC
Ford Global Tech	90	Machine Learning, Computer Memory, Strain Displacement
Qualcomm Inc	79	Output Mask, Segmentation Neural, Coordinate Information
Toyota Research Institute	55	Odometry Noise Model, Motion Sensor, Fleet-scale Datasets

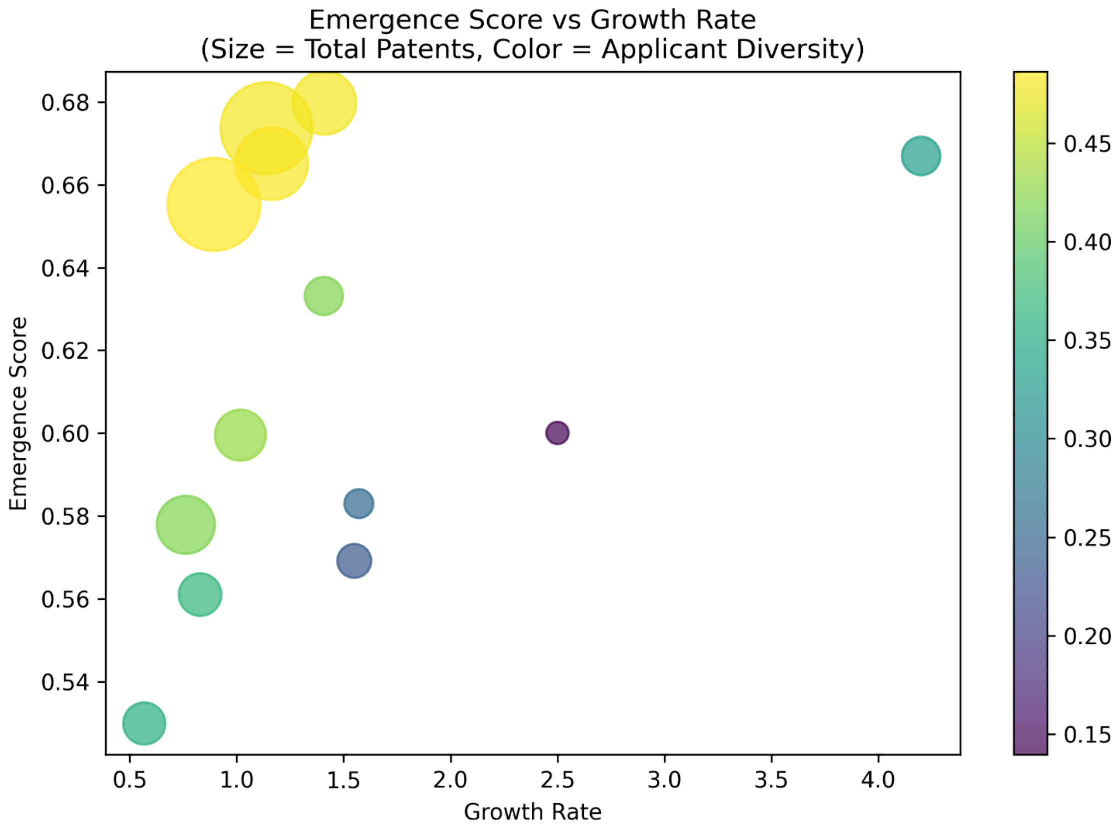
- "Computer memory" and "training data" show strongest growth (r>0.86)
- Traditional storage concepts declining as cloud/edge computing rises
- Clear shift from basic ML to specialized deep learning operations
- Intel and NVIDIA leading with 268 patents combined (40% of total)

7.3 Technology Emergence Analysis (ML)

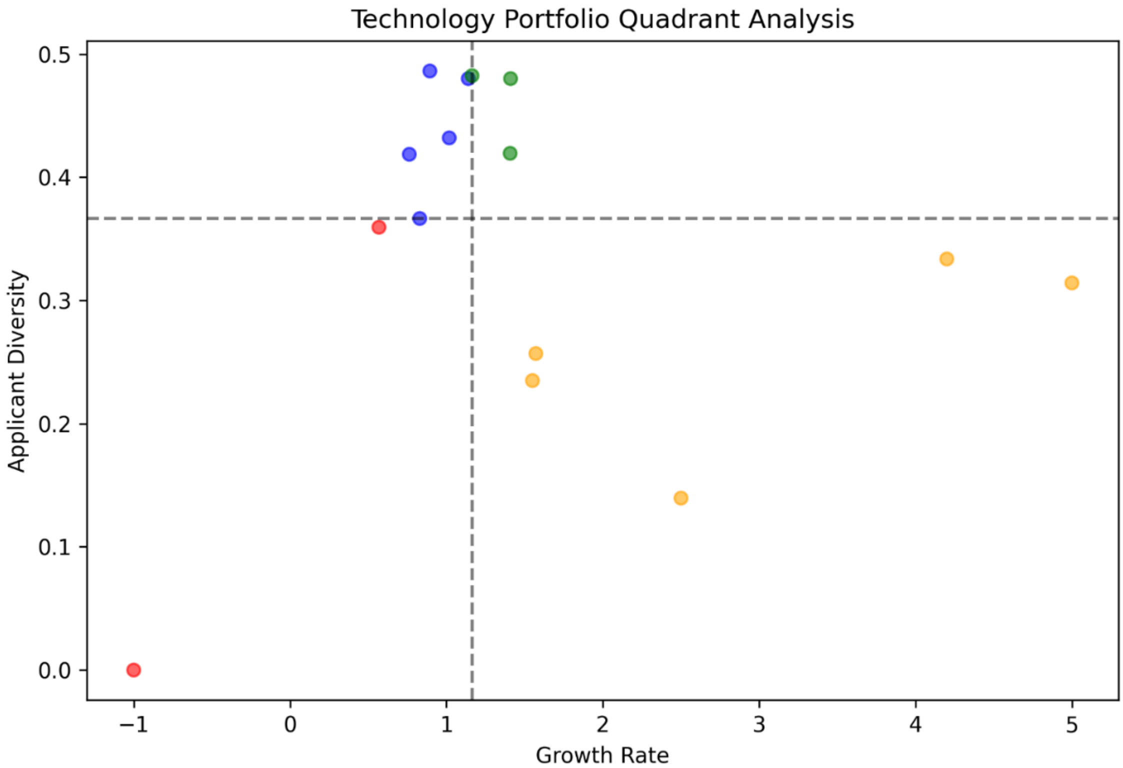
🎯 Top 5 Most Emerging AV Technologies

Rank	Technology	Emergence Score	Growth Rate	Strategic Significance
1	Perception & Sensing	0.680	1.41×	Highest emergence score - Critical for AV safety
2	Data Processing	0.674	1.14×	Second highest patents (563) + strong emergence
3	V2X Communication	0.667	4.20×	Breakthrough technology - Fastest established growth
4	Software Algorithms	0.665	1.16×	Third highest patents (349) + consistent emergence
5	AI/ML Architecture	0.655	0.89×	Most patented (574) but maturing - foundational tech

The table identifies Perception & Sensing as the most emergent and V2X Communication as the fastest-growing technology, indicating key areas of rapid advancement and strategic importance within autonomous vehicles.



The scatter plot illustrates that technologies with higher emergence scores tend to have higher growth rates, and those with more total patents (larger bubble size) and higher applicant diversity (yellowish color) generally occupy the upper-right quadrant, indicating a positive correlation between these metrics.



The Quadrant Analysis scatter plot, assessing technologies based on growth rate and applicant diversity, shows a varied distribution where some technologies demonstrate high applicant diversity, while others exhibit higher growth rates but lower diversity.

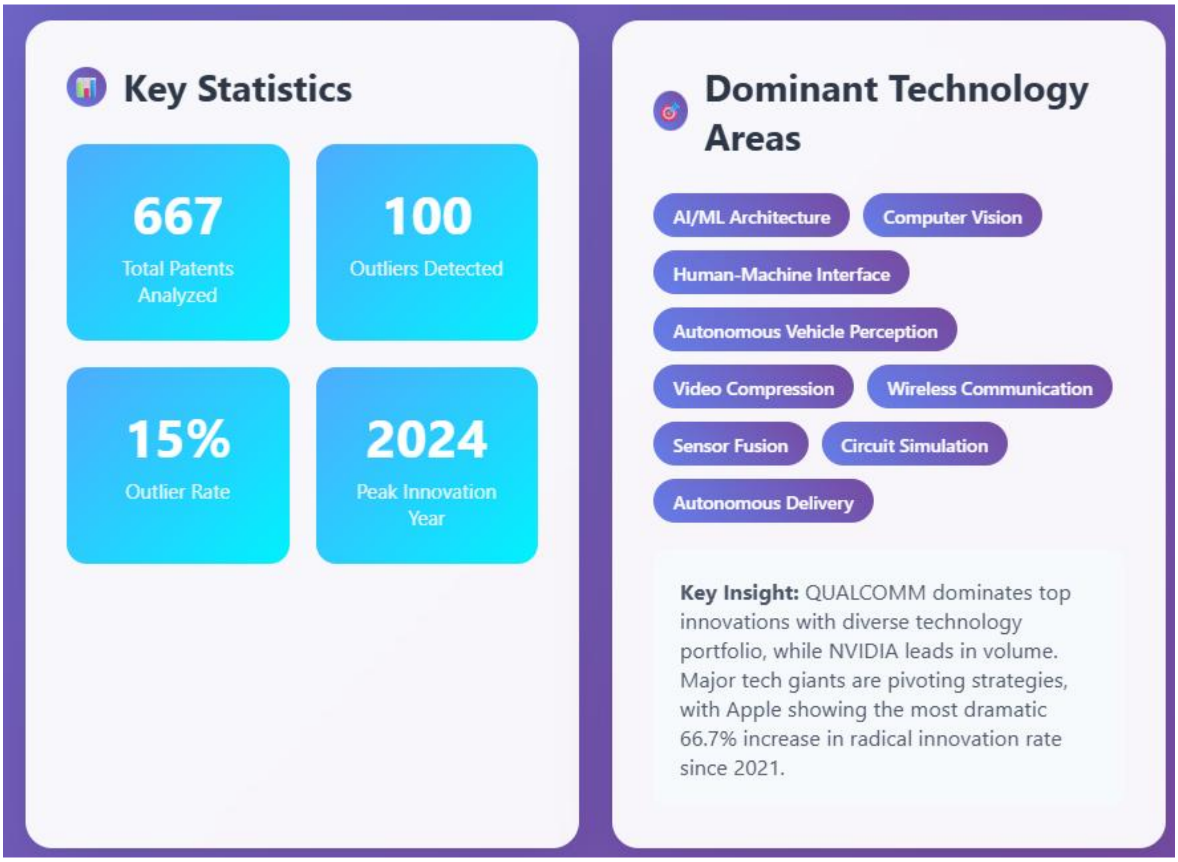
📈 Patent Volume vs. Emergence Comparison

Technology	Patent Count	Patent Rank	Emergence Rank	Gap Analysis
AI/ML Architecture	574	1	5	Mature technology - high volume, lower emergence
Data Processing	563	2	2	Balanced - high volume and emergence
Perception & Sensing	266	4	1	Emerging leader - lower volume, highest emergence
V2X Communication	94	9	3	Breakthrough - low volume, high emergence

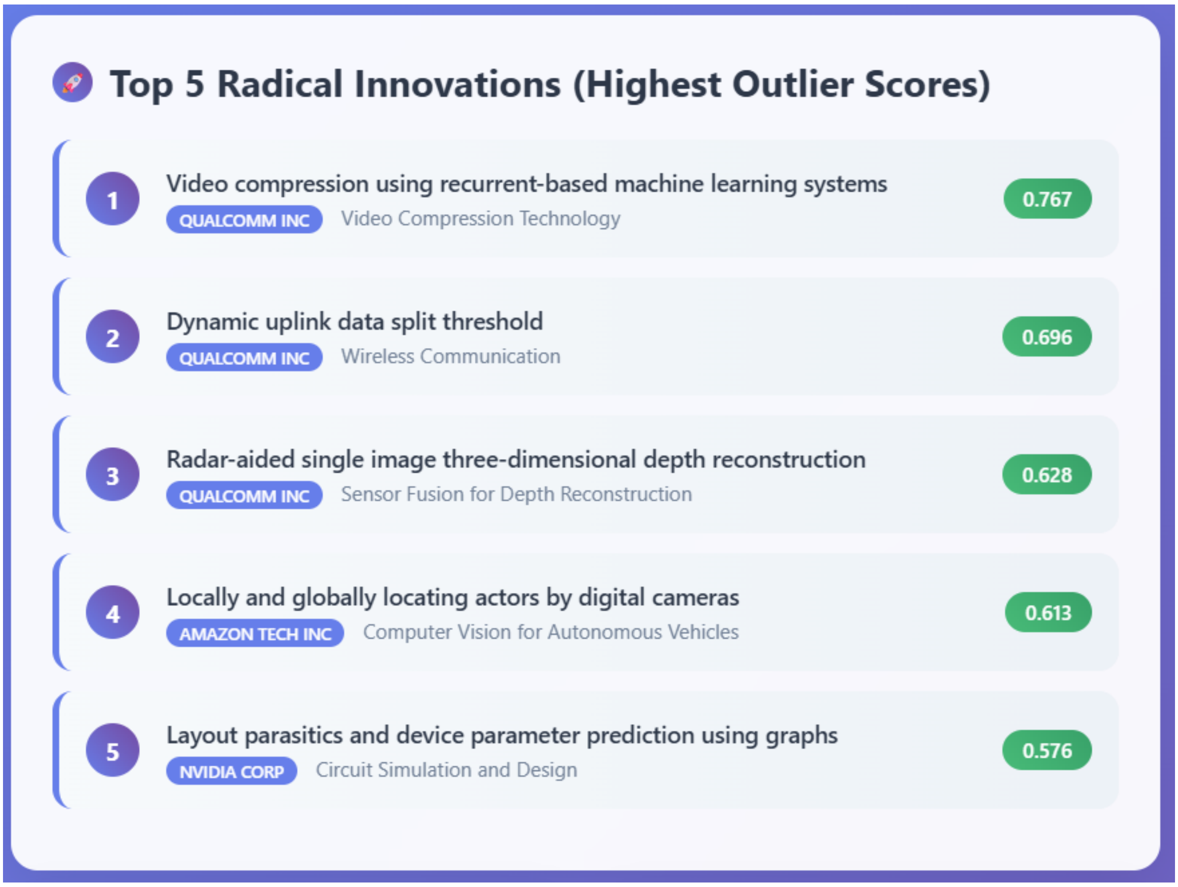
The table demonstrates a varied landscape where AI/ML Architecture represents a mature technology with high patent volume but lower emergence, while

Perception & Sensing and V2X Communication, despite lower patent counts, are identified as highly emergent and breakthrough technologies, respectively.

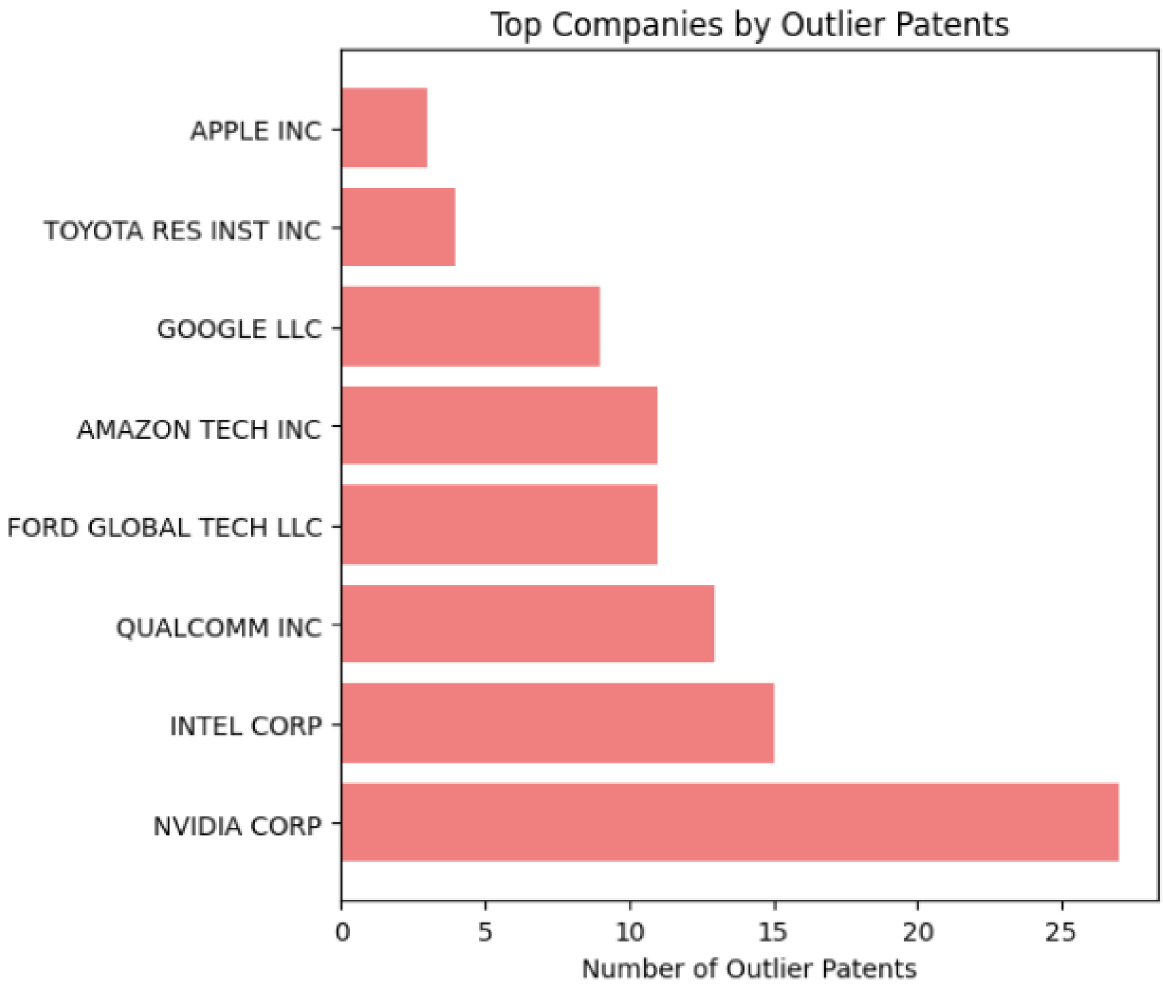
7.4 Technology Emergence Analysis (NLP + ML)



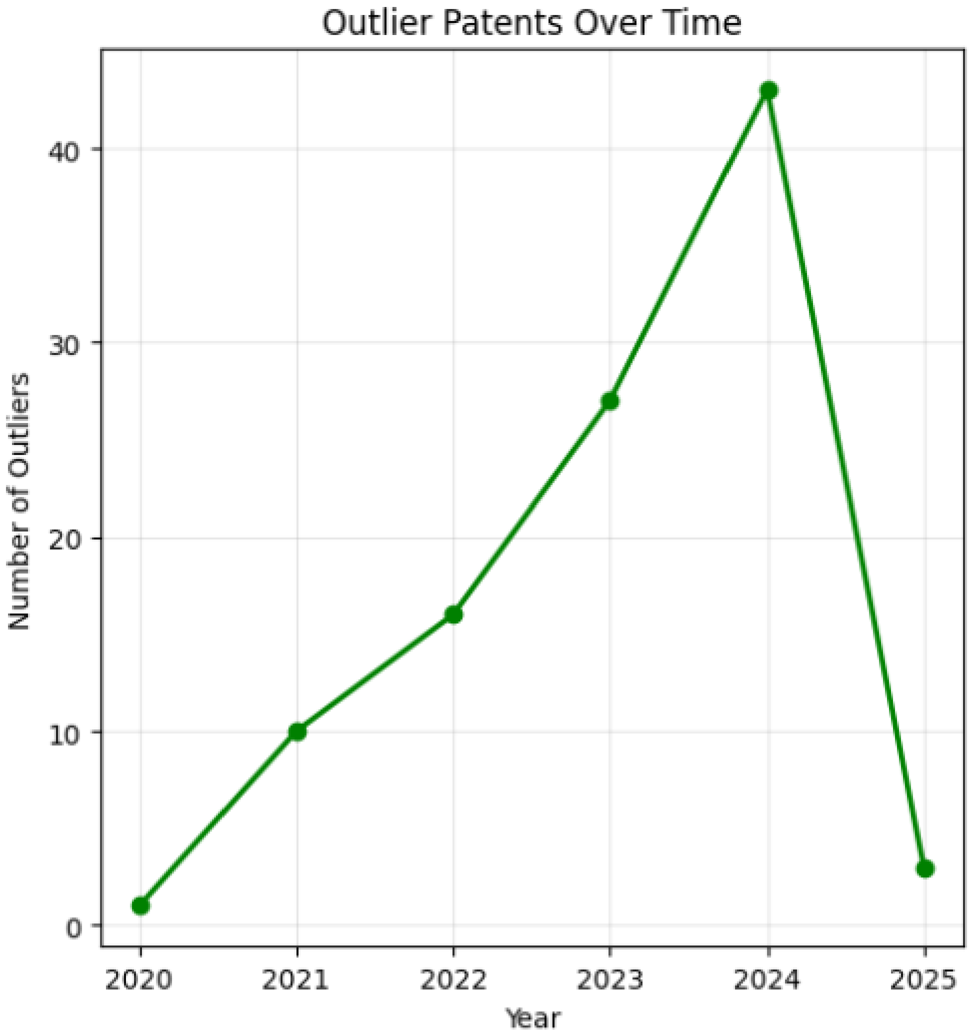
The outcome highlights that out of 667 analyzed patents, AI/ML Architecture, Computer Vision, and Autonomous Vehicle Perception are key technology areas, with 2024 being the peak innovation year, and Qualcomm and Nvidia leading in volume and strategic innovation.



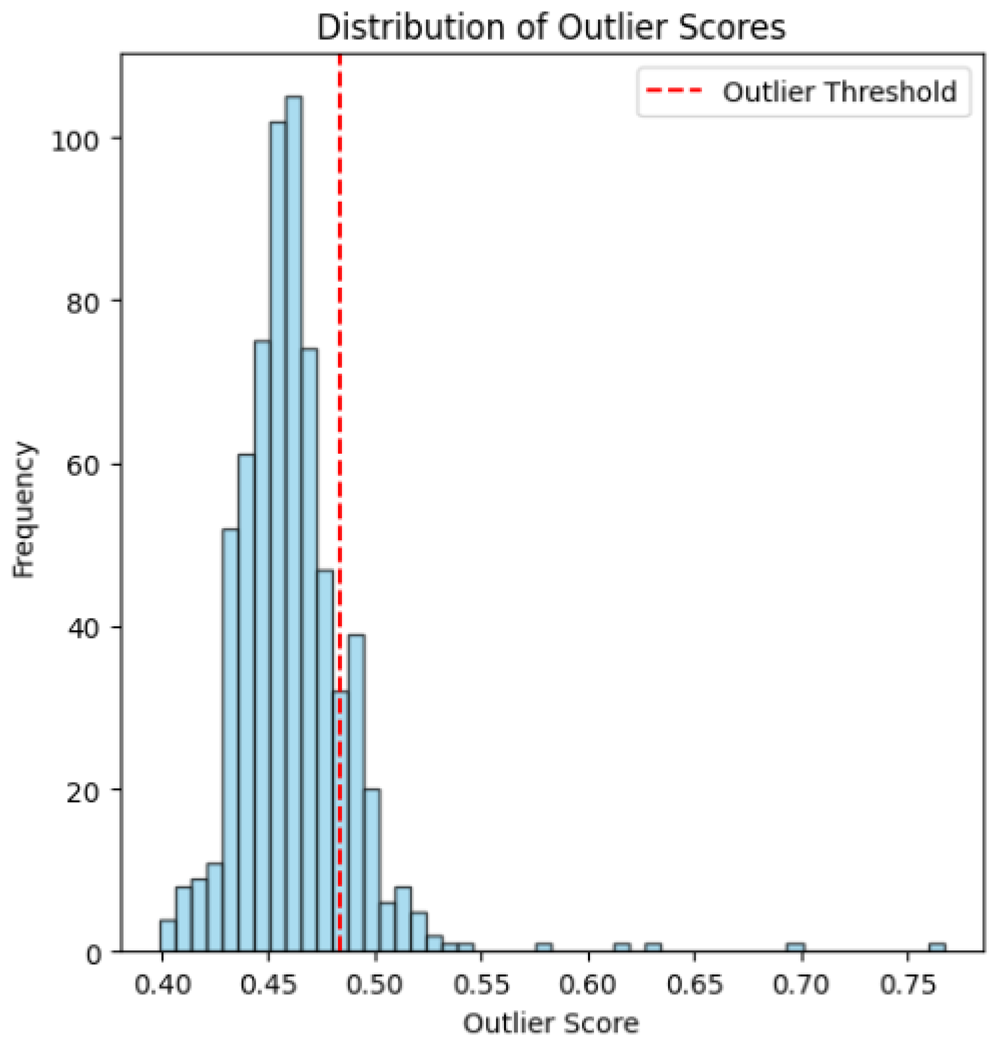
The outcome highlights Qualcomm Inc. as a dominant innovator in video compression and wireless communication, along with significant radical contributions from Amazon Tech Inc. in computer vision for autonomous vehicles and Nvidia Corp. in circuit simulation.



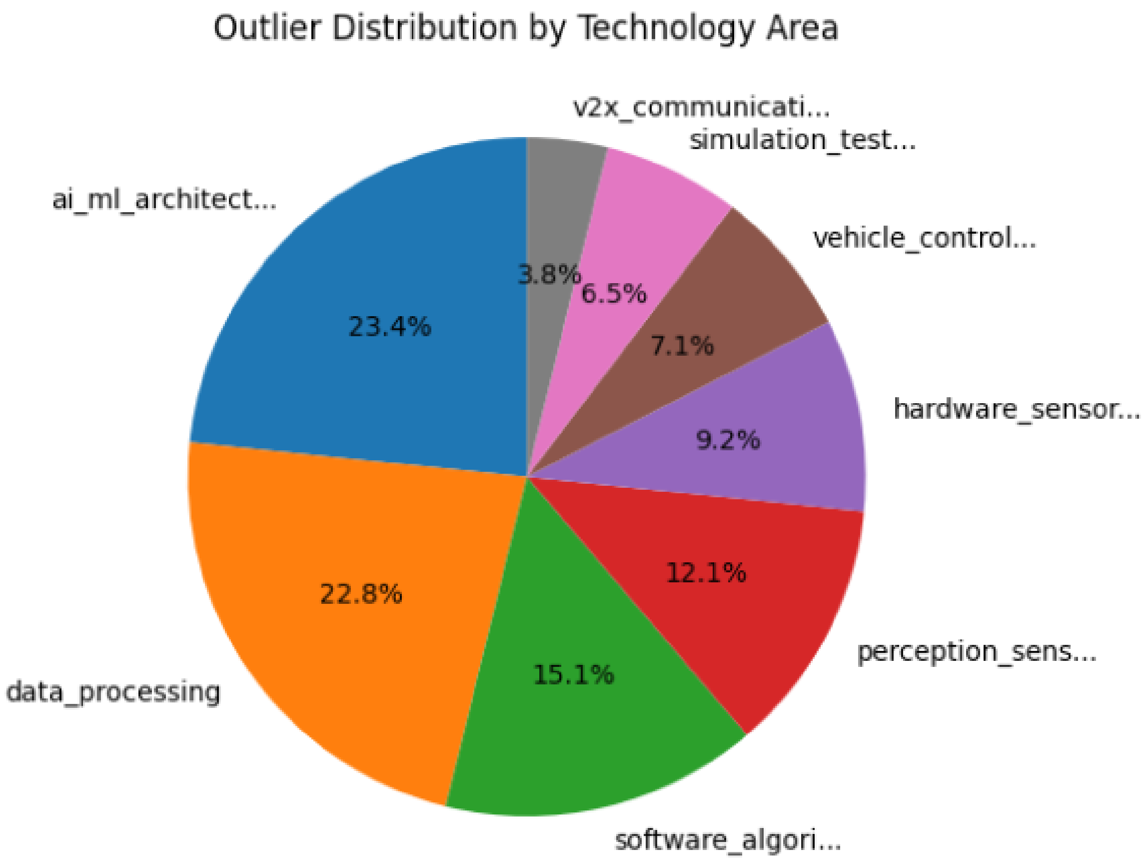
The outcome highlights Nvidia Corp. and Intel Corp. as the top patenting entities by volume, while significant increases in strategic pivots are observed for Apple Inc., Google LLC, and Amazon Tech, reflecting their evolving technological focuses.



The line chart shows a steady increase in innovation with each year with a steep drop for 2025 due to the limited number of patents in the data for the current year.



The histogram indicates that most patents have lower outlier scores, suggesting a clustering of conventional innovations, with a smaller tail of higher scores representing truly novel or "outlier" inventions exceeding the defined threshold.



The pie chart reveals that AI/ML Architecture and Data Processing account for the largest proportions of novel or "outlier" patents, indicating these are highly innovative and rapidly evolving fields within autonomous vehicle technology.

8.CONCLUSION

This comprehensive analysis of 667 US autonomous vehicle patents successfully demonstrated the power of advanced NLP and machine learning techniques in extracting strategic intelligence from large-scale patent data. Through semantic clustering using Sentence-BERT embeddings, six distinct technology clusters were identified, with Vehicle Systems emerging as the dominant area (197 patents, 29.5% of total). The analysis revealed clear market differentiation, with Intel demonstrating the broadest innovation portfolio across all clusters, while Ford and Toyota showed specialized concentrations in specific technological domains.

The innovation trend analysis uncovered significant technological shifts within the autonomous vehicle landscape. Emerging technologies like "computer memory" and "training data" showed strong growth correlations ($r>0.86$), while traditional storage concepts declined as cloud and edge computing gained prominence. This analysis captured the fundamental transition from basic machine learning approaches to specialized deep learning operations, highlighting the industry's technological evolution.

The LLM-based transformation pipeline proved highly effective in converting unstructured patent text into structured analytical data, enabling comprehensive technology emergence analysis. Key findings identified Perception & Sensing as the most emergent technology area, while V2X Communication demonstrated the fastest growth rate, indicating critical strategic investment areas for autonomous vehicle development.

Outlier detection analysis revealed that AI/ML Architecture and Data Processing accounted for the largest proportions of radical innovations, with companies like Qualcomm, Nvidia, and Amazon leading breakthrough patent development. The methodology successfully identified strategic pivots by major players including Apple, Google, and Amazon, reflecting their evolving technological focuses within the autonomous vehicle ecosystem.

This project demonstrated that sophisticated NLP methodologies, when properly implemented with robust preprocessing pipelines and multi-method validation approaches, can unlock deep, actionable intelligence from complex patent databases. The combination of semantic analysis, temporal trend detection, and emergence scoring provides stakeholders with comprehensive insights for strategic technology investment and competitive intelligence decisions in rapidly evolving markets.

9. FUTURE ENHANCEMENT

Several opportunities exist to extend and improve this patent analysis framework. **Methodological refinements** could enhance result quality by implementing advanced noise reduction techniques in semantic clustering and incorporating ensemble approaches for more robust outlier detection. **Pipeline automation** represents a critical development area, requiring the creation of end-to-end workflows that streamline data acquisition, preprocessing, analysis, and visualization processes for real-time patent monitoring capabilities.

Advanced predictive modeling should be prioritized to strengthen technology emergence analysis through sophisticated time-series forecasting and cross-domain innovation prediction models. Additional enhancements could include **multi-jurisdictional patent integration** to capture global innovation patterns, **real-time API monitoring** for continuous patent landscape updates, and **interactive dashboard development** for stakeholder engagement. Furthermore, incorporating **citation network analysis** and **inventor mobility tracking** would provide deeper insights into knowledge transfer patterns and collaborative innovation networks within the autonomous vehicle ecosystem, ultimately delivering more comprehensive competitive intelligence for strategic decision-making.

10.APPENDICIES

10.1 FULL CODE

Technology Clustering (NLP):

```
import json
import pandas as pd
import numpy as np
from sentence_transformers import SentenceTransformer
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from sklearn.metrics.pairwise import cosine_similarity
import umap
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from collections import Counter, defaultdict
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import warnings
warnings.filterwarnings('ignore')

class SemanticPatentClusterer:
    def __init__(self, jsonl_file_path, model_name='all-MiniLM-L6-v2'):
        """
        Initialize with patent data and SBERT model

        Popular model options:
        - 'all-MiniLM-L6-v2': Fast, good general performance (384 dim)
        - 'all-mpnet-base-v2': Better quality, slower (768 dim)
        - 'all-distilroberta-v1': Good balance (768 dim)
        """
        self.data = self.load_data(jsonl_file_path)
        self.model_name = model_name
        self.sbert_model = None
        self.embeddings = None
        self.cluster_labels = None
```

```
        self.processed_texts = None

        # Download required NLTK data
        try:
            nltk.data.find('tokenizers/punkt')
            nltk.data.find('corpora/stopwords')
            nltk.data.find('corpora/wordnet')
        except LookupError:
            nltk.download('punkt')
            nltk.download('stopwords')
            nltk.download('wordnet')

    def load_data(self, file_path):
        """Load and parse JSONL file"""
        data = []
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                data.append(json.loads(line.strip()))
        return pd.DataFrame(data)

    def preprocess_patent_text(self, text):
        """
        Lighter preprocessing for SBERT - preserve more semantic context
        SBERT handles context better, so we don't need aggressive
preprocessing
        """
        if pd.isna(text) or text == '':
            return ''

        # Basic cleaning while preserving technical terms
        text = re.sub(r'\b(fig\.|figure)\s*\d+\b', 'figure', text,
flags=re.IGNORECASE)
        text = re.sub(r'\bclaim\s*\d+\b', 'claim', text,
flags=re.IGNORECASE)

        # Remove excessive whitespace and special chars but keep hyphens
        text = re.sub(r'[^\w\s\-\.]', ' ', text)
        text = re.sub(r'\s+', ' ', text).strip()

        # Truncate very long texts (SBERT has token limits)
        words = text.split()
        if len(words) > 500:  # Keep to ~500 words for better processing
            text = ' '.join(words[:500])

        return text

    def prepare_semantic_corpus(self):
        """Prepare text corpus for semantic embedding"""
        print("Preparing semantic corpus...")
```

```
        # Combine claims into single text
        self.data['claims_text'] = self.data['claims'].apply(
            lambda x: ' '.join(x) if isinstance(x, list) else str(x)
        )

        # Create weighted combination prioritizing abstract and key claims
        self.data['semantic_text'] = (
            self.data['invention_title_text'].fillna('') + '. ' +
            self.data['abstract_text'].fillna('') + '. ' +
            self.data['claims_text'].fillna('')[:1000]  # Limit claims
length
        )

        # Light preprocessing
        self.processed_texts =
self.data['semantic_text'].apply(self.preprocess_patent_text)

        # Filter out very short or empty documents - FIXED: Convert to numpy
array
        valid_mask = self.processed_texts.str.len() > 100
        self.data = self.data[valid_mask].reset_index(drop=True)
        self.processed_texts =
self.processed_texts[valid_mask].reset_index(drop=True)

        # Convert to list for consistent handling
        self.processed_texts = self.processed_texts.tolist()

        print(f"Prepared {len(self.processed_texts)} valid documents for
semantic embedding")
        return self.processed_texts

    def load_sbert_model(self):
        """Load the SBERT model"""
        print(f"Loading SBERT model: {self.model_name}")
        self.sbert_model = SentenceTransformer(self.model_name)
        print(f"Model loaded. Embedding dimension:
{self.sbert_model.get_sentence_embedding_dimension()}")

    def generate_embeddings(self, batch_size=32):
        """Generate semantic embeddings for all patents"""
        if self.sbert_model is None:
            self.load_sbert_model()

        print("Generating semantic embeddings...")
        print(f"Processing {len(self.processed_texts)} documents in batches
of {batch_size}")

        # Generate embeddings in batches to manage memory
```

```
        # FIXED: Ensure processed_texts is a list
        texts_list = self.processed_texts if
isinstance(self.processed_texts, list) else self.processed_texts.tolist()

        self.embeddings = self.sbert_model.encode(
            texts_list,
            batch_size=batch_size,
            show_progress_bar=True,
            convert_to_tensor=False,
            normalize_embeddings=True  # L2 normalization for cosine
similarity
        )

        print(f"Generated embeddings shape: {self.embeddings.shape}")
        return self.embeddings

    def find_optimal_clusters_semantic(self, max_k=20, sample_size=None):
        """Find optimal number of clusters using matplotlib visualization"""
        print("Finding optimal number of clusters using semantic
embeddings...")

        embeddings_array = np.array(self.embeddings)

        if sample_size and len(embeddings_array) > sample_size:
            indices = np.random.choice(len(embeddings_array), sample_size,
replace=False)
            sample_embeddings = embeddings_array[indices]
            print(f"Using sample of {sample_size} documents for
optimization")
        else:
            sample_embeddings = embeddings_array

        k_range = range(5, min(max_k + 1, len(sample_embeddings) // 10))
        silhouette_scores = []
        calinski_scores = []
        inertias = []

        for k in k_range:
            print(f"Testing k={k}...")
            try:
                kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
                labels = kmeans.fit_predict(sample_embeddings)

                if len(np.unique(labels)) > 1:
                    sil_score = silhouette_score(sample_embeddings, labels)
                    cal_score = calinski_harabasz_score(sample_embeddings,
labels)
                else:
                    sil_score = -1
```

```
                cal_score = 0

            silhouette_scores.append(sil_score)
            calinski_scores.append(cal_score)
            inertias.append(kmeans.inertia_)

        except Exception as e:
            print(f"Error with k={k}: {e}")
            silhouette_scores.append(-1)
            calinski_scores.append(0)
            inertias.append(float('inf'))

    # Create matplotlib visualization
    try:
        fig, axes = plt.subplots(1, 3, figsize=(18, 5))

        # Silhouette Score
        axes[0].plot(k_range, silhouette_scores, 'bo-', linewidth=2,
markersize=8)
        axes[0].set_title('Silhouette Score', fontsize=14,
fontweight='bold')
        axes[0].set_xlabel('Number of Clusters', fontsize=12)
        axes[0].set_ylabel('Silhouette Score', fontsize=12)
        axes[0].grid(True, alpha=0.3)
        axes[0].tick_params(labelsize=10)

        # Highlight the best score
        if max(silhouette_scores) > -1:
            best_k_sil = k_range[np.argmax(silhouette_scores)]
            axes[0].axvline(x=best_k_sil, color='red', linestyle='--',
alpha=0.7)
            axes[0].text(best_k_sil, max(silhouette_scores), f'Best:
{best_k_sil}',
                       ha='center', va='bottom', fontweight='bold',
color='red')

        # Calinski-Harabasz Score
        axes[1].plot(k_range, calinski_scores, 'ro-', linewidth=2,
markersize=8)
        axes[1].set_title('Calinski-Harabasz Score', fontsize=14,
fontweight='bold')
        axes[1].set_xlabel('Number of Clusters', fontsize=12)
        axes[1].set_ylabel('Calinski-Harabasz Score', fontsize=12)
        axes[1].grid(True, alpha=0.3)
        axes[1].tick_params(labelsize=10)

        # Highlight the best score
        if max(calinski_scores) > 0:
            best_k_cal = k_range[np.argmax(calinski_scores)]
```

```
                axes[1].axvline(x=best_k_cal, color='red', linestyle='--',
alpha=0.7)
                axes[1].text(best_k_cal, max(calinski_scores), f'Best:
{best_k_cal}',
                           ha='center', va='bottom', fontweight='bold',
color='red')

            # Inertia (Elbow Method)
            axes[2].plot(k_range, inertias, 'go-', linewidth=2,
markersize=8)
            axes[2].set_title('Inertia (Elbow Method)', fontsize=14,
fontweight='bold')
            axes[2].set_xlabel('Number of Clusters', fontsize=12)
            axes[2].set_ylabel('Inertia', fontsize=12)
            axes[2].grid(True, alpha=0.3)
            axes[2].tick_params(labelsize=10)

            # Find elbow point (simple method)
            if len(inertias) > 2:
                # Calculate the rate of change
                differences = np.diff(inertias)
                if len(differences) > 1:
                    second_diff = np.diff(differences)
                    if len(second_diff) > 0:
                        elbow_idx = np.argmax(second_diff) + 2  # +2 because
of double diff
                        if elbow_idx < len(k_range):
                            elbow_k = k_range[elbow_idx]
                            axes[2].axvline(x=elbow_k, color='red',
linestyle='--', alpha=0.7)
                            axes[2].text(elbow_k, inertias[elbow_idx],
f'Elbow: {elbow_k}',
                                       ha='center', va='bottom',
fontweight='bold', color='red')

            plt.suptitle('Clustering Optimization Metrics', fontsize=16,
fontweight='bold')
            plt.tight_layout()
            plt.savefig('clustering_optimization_metrics.png', dpi=300,
bbox_inches='tight')
            plt.show()
            print("Matplotlib clustering optimization plot: SUCCESS")

        except Exception as e:
            print(f"Matplotlib plotting failed: {e}")

        # Return optimal k
        valid_scores = [s for s in silhouette_scores if s > -1]
        if valid_scores:
```

```
            optimal_k = k_range[np.argmax(silhouette_scores)]
            print(f"Optimal number of clusters: {optimal_k} (Silhouette
score: {max(silhouette_scores):.3f})")
        else:
            optimal_k = 8
            print(f"Using default clusters: {optimal_k}")

        return optimal_k, silhouette_scores, calinski_scores

    def perform_semantic_clustering(self, n_clusters=None, method='kmeans'):
        """Perform clustering on semantic embeddings"""
        if n_clusters is None:
            n_clusters, _, _ = self.find_optimal_clusters_semantic()

        print(f"Performing semantic {method} clustering with {n_clusters}
clusters...")

        # Ensure embeddings are proper numpy array
        embeddings_array = np.array(self.embeddings)

        if method == 'kmeans':
            clusterer = KMeans(n_clusters=n_clusters, random_state=42,
n_init=10)
        elif method == 'hierarchical':
            clusterer = AgglomerativeClustering(n_clusters=n_clusters,
linkage='ward')
        elif method == 'dbscan':
            # For semantic embeddings, use cosine distance
            clusterer = DBSCAN(eps=0.3, min_samples=5, metric='cosine')

        # Fit and predict
        try:
            self.cluster_labels = clusterer.fit_predict(embeddings_array)
        except Exception as e:
            print(f"Clustering failed with error: {e}")
            print("Falling back to simple KMeans...")
            clusterer = KMeans(n_clusters=min(n_clusters, 10),
random_state=42, n_init=10)
            self.cluster_labels = clusterer.fit_predict(embeddings_array)

        # FIXED: Properly assign cluster labels to dataframe
        # Ensure we have the right number of labels for our filtered data
        if len(self.cluster_labels) == len(self.data):
            self.data = self.data.copy()
            self.data['cluster'] = self.cluster_labels
        else:
            print(f"Warning: Mismatch between cluster labels
({len(self.cluster_labels)}) and data ({len(self.data)})")
            # Truncate or pad as needed
```

```
            if len(self.cluster_labels) > len(self.data):
                self.data['cluster'] = self.cluster_labels[:len(self.data)]
            else:
                # This shouldn't happen if filtering was done correctly
                extended_labels = np.concatenate([self.cluster_labels,
np.full(len(self.data) - len(self.cluster_labels), -1)])
                self.data['cluster'] = extended_labels

        # Calculate clustering quality
        unique_labels = np.unique(self.cluster_labels)
        if len(unique_labels) > 1 and -1 not in unique_labels:
            try:
                sil_score = silhouette_score(embeddings_array,
self.cluster_labels)
                print(f"Clustering silhouette score: {sil_score:.3f}")
            except Exception as e:
                print(f"Could not calculate silhouette score: {e}")

        # Print cluster distribution
        cluster_counts = pd.value_counts(self.cluster_labels, sort=False)
        print("Cluster distribution:")
        for cluster_id in sorted(cluster_counts.index):
            count = cluster_counts[cluster_id]
            print(f"  Cluster {cluster_id}: {count} patents")

        return clusterer

    def extract_semantic_cluster_keywords(self, top_k=15):
        """Extract representative keywords using semantic centroids and TF-
IDF"""
        print("Extracting semantic cluster keywords...")

        from sklearn.feature_extraction.text import TfidfVectorizer

        # FIXED: Ensure processed_texts is a list for TfidfVectorizer
        texts_list = self.processed_texts if
isinstance(self.processed_texts, list) else self.processed_texts.tolist()

        # Create TF-IDF for keyword extraction
        vectorizer = TfidfVectorizer(
            max_features=3000,
            ngram_range=(1, 3),
            min_df=2,
            max_df=0.8,
            stop_words='english'
        )

        tfidf_matrix = vectorizer.fit_transform(texts_list)
        feature_names = vectorizer.get_feature_names_out()
```

```
        cluster_keywords = {}
        cluster_centroids = {}

        for cluster_id in sorted(self.data['cluster'].unique()):
            if cluster_id == -1:  # Skip noise cluster
                continue

            # FIXED: Use numpy array indexing instead of pandas boolean
indexing
            cluster_mask = (self.data['cluster'] == cluster_id).values
            cluster_embeddings = self.embeddings[cluster_mask]
            cluster_tfidf = tfidf_matrix[cluster_mask]

            # Calculate semantic centroid
            centroid = np.mean(cluster_embeddings, axis=0)
            cluster_centroids[cluster_id] = centroid

            # Get TF-IDF scores for cluster documents
            cluster_tfidf_mean =
np.array(cluster_tfidf.mean(axis=0)).flatten()

            # Get top terms by TF-IDF
            top_indices = cluster_tfidf_mean.argsort()[-top_k:][::-1]
            top_terms = [(feature_names[i], cluster_tfidf_mean[i]) for i in
top_indices]

            cluster_keywords[cluster_id] = top_terms

        self.cluster_centroids = cluster_centroids
        return cluster_keywords

    def create_semantic_visualization(self, method='umap', n_components=2):
        """Create visualization using matplotlib only"""
        print(f"Creating {method.upper()} visualization...")

        embeddings_array = np.array(self.embeddings)

        # Dimensionality reduction with better error handling
        coords = None
        reduction_success = False

        try:
            if method == 'umap' and umap is not None:
                reducer = umap.UMAP(n_components=n_components,
n_neighbors=15,
                                  min_dist=0.1, metric='cosine',
random_state=42)
            elif method == 'tsne':
```

```
                reducer = TSNE(n_components=n_components,
                              perplexity=min(30, len(embeddings_array) //
4),
                              random_state=42, metric='cosine')
            else:
                reducer = PCA(n_components=n_components, random_state=42)

            coords = reducer.fit_transform(embeddings_array)
            reduction_success = True

        except Exception as e:
            print(f"Error with {method}: {e}")
            print("Falling back to PCA...")
            try:
                reducer = PCA(n_components=n_components, random_state=42)
                coords = reducer.fit_transform(embeddings_array)
                reduction_success = True
                method = 'PCA'  # Update method name for plot title
            except Exception as e:
                print(f"PCA also failed: {e}")
                return None

        if not reduction_success or coords is None:
            print("All dimensionality reduction methods failed!")
            return None

        # Create matplotlib visualization
        if n_components == 2:
            try:
                # Create a larger, more detailed plot
                fig, ax = plt.subplots(figsize=(14, 10))

                # Get unique clusters and create a color map
                unique_clusters = np.unique(self.cluster_labels)
                colors = plt.cm.tab20(np.linspace(0, 1,
len(unique_clusters)))

                # Plot each cluster with different colors
                for i, cluster_id in enumerate(unique_clusters):
                    cluster_mask = self.cluster_labels == cluster_id
                    cluster_coords = coords[cluster_mask]

                    if cluster_id == -1:  # Noise cluster
                        ax.scatter(cluster_coords[:, 0], cluster_coords[:,
1],
                                 c='gray', alpha=0.5, s=30,
edgecolors='black',
                                 linewidth=0.5, label=f'Noise', marker='x')
                    else:
```

```
                        ax.scatter(cluster_coords[:, 0], cluster_coords[:,
1],
                                 c=[colors[i]], alpha=0.7, s=60,
edgecolors='black',
                                 linewidth=0.5, label=f'Cluster
{cluster_id}')

                # Customize the plot
                ax.set_title(f'Semantic Patent Clusters ({method.upper()}
Visualization)',
                           fontsize=16, fontweight='bold', pad=20)
                ax.set_xlabel(f'{method.upper()} Dimension 1', fontsize=12)
                ax.set_ylabel(f'{method.upper()} Dimension 2', fontsize=12)
                ax.grid(True, alpha=0.3)
                ax.tick_params(labelsize=10)

                # Add legend with better positioning
                if len(unique_clusters) <= 20:  # Only show legend if not
too many clusters
                    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left',
fontsize=10)

                # Add cluster centroids
                if hasattr(self, 'cluster_centroids') and
self.cluster_centroids:
                    for cluster_id in self.cluster_centroids.keys():
                        if cluster_id != -1:
                            cluster_mask = self.cluster_labels == cluster_id
                            cluster_coords = coords[cluster_mask]
                            if len(cluster_coords) > 0:
                                centroid_x = np.mean(cluster_coords[:, 0])
                                centroid_y = np.mean(cluster_coords[:, 1])
                                ax.scatter(centroid_x, centroid_y, c='red',
s=200,
                                         marker='*', edgecolors='black',
linewidth=2,
                                         alpha=0.8, zorder=5)
                                ax.text(centroid_x, centroid_y,
str(cluster_id),
                                       ha='center', va='center',
fontweight='bold',
                                       fontsize=10, color='white')

                plt.tight_layout()
                plt.savefig('semantic_cluster_visualization.png', dpi=300,
bbox_inches='tight')
                plt.show()
                print("Matplotlib cluster visualization: SUCCESS")
```

```
                # Additional detailed view for clusters
                self._create_cluster_detail_plot(coords, method)

            except Exception as e:
                print(f"Matplotlib visualization failed: {e}")
                return None

        return coords

    def _create_cluster_detail_plot(self, coords, method):
        """Create additional detailed cluster plots with applicant
analysis"""
        try:
            # Ensure we have applicant analysis
            if not hasattr(self, 'cluster_applicants'):
                self.analyze_cluster_applicants()

            unique_clusters = np.unique(self.cluster_labels)
            valid_clusters = [c for c in unique_clusters if c != -1]

            if len(valid_clusters) <= 16:
                # Determine number of columns based on actual cluster count
                n_cols = min(len(valid_clusters), 8)  # Cap at 8 for
readability

                # Create two sets of subplots: regular clusters and
applicant-colored clusters
                fig, axes = plt.subplots(2, n_cols, figsize=(4*n_cols, 8))

                # Handle single column case
                if n_cols == 1:
                    axes = axes.reshape(2, 1)

                # Show all clusters (not just first 4)
                clusters_to_show = valid_clusters[:n_cols]
                colors = plt.cm.tab20(np.linspace(0, 1,
len(clusters_to_show)))

                for i, cluster_id in enumerate(clusters_to_show):
                    # Row 1: Regular cluster view
                    ax1 = axes[0, i] if n_cols > 1 else axes[0]

                    # Plot all points in gray
                    ax1.scatter(coords[:, 0], coords[:, 1], c='lightgray',
                              alpha=0.3, s=20, edgecolors='none')

                    # Highlight current cluster
                    cluster_mask = self.cluster_labels == cluster_id
                    cluster_coords = coords[cluster_mask]
```

```
                    ax1.scatter(cluster_coords[:, 0], cluster_coords[:, 1],
                              c=[colors[i]], alpha=0.8, s=40,
edgecolors='black',
                              linewidth=0.5)

                    ax1.set_title(f'Cluster {cluster_id}
({np.sum(cluster_mask)} patents)',
                                fontsize=10, fontweight='bold')
                    ax1.grid(True, alpha=0.3)
                    ax1.tick_params(labelsize=8)

                    # Row 2: Applicant-colored view
                    ax2 = axes[1, i] if n_cols > 1 else axes[1]

                    # Plot all points in gray
                    ax2.scatter(coords[:, 0], coords[:, 1], c='lightgray',
                              alpha=0.3, s=20, edgecolors='none')

                    # Get cluster data and top applicants
                    cluster_data = self.data[self.data['cluster'] ==
cluster_id]
                    top_applicants =
list(self.cluster_applicants[cluster_id]['top_applicants'].keys())[:5]

                    # Create color map for top applicants
                    applicant_colors = plt.cm.Set3(np.linspace(0, 1,
len(top_applicants)))

                    # Plot each top applicant with different color
                    for j, applicant in enumerate(top_applicants):
                        applicant_mask = (cluster_data['applicant_name'] ==
applicant).values
                        if np.any(applicant_mask):
                            # Get indices in the original data
                            cluster_indices =
cluster_data.index[applicant_mask]
                            # Map back to coordinates
                            coord_indices = [idx for idx in
range(len(self.data))
                                          if self.data.index[idx] in
cluster_indices]

                            if coord_indices:
                                applicant_coords = coords[coord_indices]
                                ax2.scatter(applicant_coords[:, 0],
applicant_coords[:, 1],
                                          c=[applicant_colors[j]],
alpha=0.8, s=50,
                                          edgecolors='black', linewidth=0.5,
```

```
                                            label=f'{applicant[:20]}...' if
len(applicant) > 20 else applicant)

                        ax2.set_title(f'Cluster {cluster_id} - Top Applicants',
                                    fontsize=10, fontweight='bold')
                        ax2.grid(True, alpha=0.3)
                        ax2.tick_params(labelsize=8)

                        # Add legend for applicants (small font)
                        if len(top_applicants) > 0:
                            ax2.legend(fontsize=6, loc='upper right',
bbox_to_anchor=(1, 1))

                    # Hide empty subplots if needed
                    for i in range(len(clusters_to_show), n_cols):
                        if n_cols > 1:
                            axes[0, i].set_visible(False)
                            axes[1, i].set_visible(False)

                    plt.suptitle(f'Cluster Views: Technology Clusters (top) vs
Applicant Distribution (bottom)',
                                fontsize=14, fontweight='bold')
                    plt.tight_layout()
                    plt.savefig('cluster_detail_with_applicants.png', dpi=300,
bbox_inches='tight')
                    plt.show()
                    print("Cluster detail with applicant visualization:
SUCCESS")

            except Exception as e:
                print(f"Failed to create cluster detail plot with applicants:
{e}")

    def create_applicant_cluster_heatmap(self):
        """Create heatmap showing applicant distribution across clusters"""
        try:
            # Get top applicants overall
            top_global_applicants =
self.data['applicant_name'].value_counts().head(15).index.tolist()

            # Create matrix: clusters x applicants
            cluster_ids = sorted([c for c in self.data['cluster'].unique()
if c != -1])

            heatmap_data = []
            for cluster_id in cluster_ids:
                cluster_data = self.data[self.data['cluster'] == cluster_id]
                row = []
                for applicant in top_global_applicants:
```

```
                    count = len(cluster_data[cluster_data['applicant_name']
== applicant])
                    row.append(count)
                heatmap_data.append(row)

            heatmap_array = np.array(heatmap_data)

            # Create heatmap
            fig, ax = plt.subplots(figsize=(16, 8))

            im = ax.imshow(heatmap_array, cmap='YlOrRd', aspect='auto')

            # Add colorbar
            cbar = plt.colorbar(im, ax=ax)
            cbar.set_label('Number of Patents', fontsize=12)

            # Set ticks and labels
            ax.set_xticks(range(len(top_global_applicants)))
            ax.set_yticks(range(len(cluster_ids)))

            # Truncate long applicant names
            truncated_applicants = [name[:25] + '...' if len(name) > 25 else
name
                                   for name in top_global_applicants]

            ax.set_xticklabels(truncated_applicants, rotation=45,
ha='right', fontsize=9)
            ax.set_yticklabels([f'Cluster {cid}' for cid in cluster_ids],
fontsize=10)

            # Add text annotations for non-zero values
            for i in range(len(cluster_ids)):
                for j in range(len(top_global_applicants)):
                    if heatmap_array[i, j] > 0:
                        text = ax.text(j, i, str(heatmap_array[i, j]),
                                     ha="center", va="center",
                                     color="white" if heatmap_array[i, j] >
heatmap_array.max()/2 else "black",
                                     fontsize=8, fontweight='bold')

            ax.set_title('Applicant Distribution Across Technology
Clusters',
                        fontsize=14, fontweight='bold', pad=20)
            ax.set_xlabel('Top Applicants', fontsize=12)
            ax.set_ylabel('Technology Clusters', fontsize=12)

            plt.tight_layout()
            plt.savefig('applicant_cluster_heatmap.png', dpi=300,
bbox_inches='tight')
```

```
            plt.show()
            print("Applicant-cluster heatmap: SUCCESS")

            return heatmap_array, cluster_ids, top_global_applicants

        except Exception as e:
            print(f"Failed to create applicant cluster heatmap: {e}")
            return None, None, None

    def find_similar_patents(self, patent_idx, top_k=10):
        """Find most similar patents using semantic similarity"""
        if self.embeddings is None:
            print("Embeddings not generated yet!")
            return None

        target_embedding = self.embeddings[patent_idx].reshape(1, -1)
        similarities = cosine_similarity(target_embedding,
self.embeddings)[0]

        # Get top k similar patents (excluding the patent itself)
        top_indices = similarities.argsort()[-top_k-1:-1][::-1]

        similar_patents = []
        for idx in top_indices:
            similar_patents.append({
                'index': idx,
                'lens_id': self.data.iloc[idx]['lens_id'],
                'title': self.data.iloc[idx]['invention_title_text'],
                'applicant': self.data.iloc[idx]['applicant_name'],
                'similarity': similarities[idx],
                'cluster': self.data.iloc[idx]['cluster']
            })

        return similar_patents

    def analyze_cluster_applicants(self, top_n=5):
        """Analyze top applicants for each cluster"""
        print("Analyzing cluster applicants...")

        cluster_applicants = {}

        for cluster_id in sorted(self.data['cluster'].unique()):
            if cluster_id == -1:
                continue

            cluster_data = self.data[self.data['cluster'] == cluster_id]

            # Get top applicants in this cluster
```

```
            applicant_counts =
cluster_data['applicant_name'].value_counts().head(top_n)

            cluster_applicants[cluster_id] = {
                'top_applicants': applicant_counts.to_dict(),
                'total_patents': len(cluster_data),
                'unique_applicants':
cluster_data['applicant_name'].nunique()
            }

        self.cluster_applicants = cluster_applicants
        return cluster_applicants

    def analyze_cluster_semantic_coherence(self):
        """Analyze semantic coherence within clusters"""
        print("Analyzing cluster semantic coherence...")

        coherence_scores = {}

        for cluster_id in sorted(self.data['cluster'].unique()):
            if cluster_id == -1:
                continue

            # FIXED: Use numpy array indexing
            cluster_mask = (self.data['cluster'] == cluster_id).values
            cluster_embeddings = self.embeddings[cluster_mask]

            if len(cluster_embeddings) < 2:
                coherence_scores[cluster_id] = 0.0
                continue

            # Calculate pairwise cosine similarities within cluster
            similarities = cosine_similarity(cluster_embeddings)

            # Get upper triangle (excluding diagonal)
            upper_triangle = similarities[np.triu_indices_from(similarities,
k=1)]

            # Average similarity is coherence score
            coherence_scores[cluster_id] = np.mean(upper_triangle)

        return coherence_scores

    def cross_cluster_similarity_analysis(self):
        """Analyze similarity between different clusters using matplotlib"""
        print("Analyzing cross-cluster similarities...")

        if not hasattr(self, 'cluster_centroids'):
            print("Need to run extract_semantic_cluster_keywords first!")
```

```
                return None

            cluster_ids = sorted([cid for cid in self.cluster_centroids.keys()
if cid != -1])
            n_clusters = len(cluster_ids)

            # Calculate centroid similarities
            similarity_matrix = np.zeros((n_clusters, n_clusters))

            for i, cluster_i in enumerate(cluster_ids):
                for j, cluster_j in enumerate(cluster_ids):
                    centroid_i = self.cluster_centroids[cluster_i].reshape(1, -
1)
                    centroid_j = self.cluster_centroids[cluster_j].reshape(1, -
1)
                    similarity_matrix[i, j] = cosine_similarity(centroid_i,
centroid_j)[0, 0]

            # Create matplotlib heatmap
            try:
                fig, ax = plt.subplots(figsize=(12, 10))

                # Create heatmap
                im = ax.imshow(similarity_matrix, cmap='viridis', aspect='auto')

                # Add colorbar
                cbar = plt.colorbar(im, ax=ax)
                cbar.set_label('Cosine Similarity', fontsize=12)

                # Set ticks and labels
                ax.set_xticks(range(n_clusters))
                ax.set_yticks(range(n_clusters))
                ax.set_xticklabels([f'Cluster {cid}' for cid in cluster_ids],
                                   rotation=45, ha='right')
                ax.set_yticklabels([f'Cluster {cid}' for cid in cluster_ids])

                # Add text annotations
                for i in range(n_clusters):
                    for j in range(n_clusters):
                        text = ax.text(j, i, f'{similarity_matrix[i, j]:.3f}',
                                    ha="center", va="center", color="white" if
similarity_matrix[i, j] < 0.5 else "black",
                                    fontsize=8)

                ax.set_title('Cross-Cluster Semantic Similarity Matrix',
                            fontsize=14, fontweight='bold', pad=20)
                plt.tight_layout()
                plt.savefig('cross_cluster_similarity_matrix.png', dpi=300,
bbox_inches='tight')
```

```
            plt.show()
            print("Cross-cluster similarity matrix: SUCCESS")

        except Exception as e:
            print(f"Failed to create similarity matrix plot: {e}")

        return similarity_matrix, cluster_ids

    def run_complete_semantic_analysis(self, n_clusters=None,
visualization_method='umap'):
        """Run complete semantic clustering analysis with applicant
insights"""
        print("=== Starting Complete Semantic Patent Technology Clustering
===\n")

        # Step 1: Prepare corpus
        self.prepare_semantic_corpus()

        # Step 2: Load SBERT model and generate embeddings
        self.load_sbert_model()
        self.generate_embeddings()

        # Step 3: Find optimal clusters and perform clustering
        clusterer = self.perform_semantic_clustering(n_clusters=n_clusters)

        # Step 4: Extract cluster keywords
        cluster_keywords = self.extract_semantic_cluster_keywords()

        # Step 5: Analyze applicants
        cluster_applicants = self.analyze_cluster_applicants()

        # Step 6: Create visualizations (now includes applicant views)
        coords =
self.create_semantic_visualization(method=visualization_method)

        # Step 7: Create applicant heatmap
        heatmap_data, cluster_ids, top_applicants =
self.create_applicant_cluster_heatmap()

        # Step 8: Analyze cluster coherence
        coherence_scores = self.analyze_cluster_semantic_coherence()

        # Step 9: Cross-cluster analysis
        similarity_matrix, cluster_ids =
self.cross_cluster_similarity_analysis()

        # Step 10: Generate comprehensive report (now includes applicant
analysis)
        self.generate_semantic_report(cluster_keywords, coherence_scores)
```

```
        return {
            'clusterer': clusterer,
            'embeddings': self.embeddings,
            'cluster_keywords': cluster_keywords,
            'cluster_applicants': cluster_applicants,
            'coordinates': coords,
            'coherence_scores': coherence_scores,
            'similarity_matrix': similarity_matrix,
            'applicant_heatmap': heatmap_data
        }

    def generate_semantic_report(self, cluster_keywords, coherence_scores):
        """Generate comprehensive semantic clustering report with applicant
analysis"""
        print("\n=== SEMANTIC PATENT CLUSTERING REPORT ===\n")

        # Ensure we have applicant analysis
        if not hasattr(self, 'cluster_applicants'):
            self.analyze_cluster_applicants()

        for cluster_id in sorted(cluster_keywords.keys()):
            cluster_data = self.data[self.data['cluster'] == cluster_id]
            keywords = cluster_keywords[cluster_id]
            coherence = coherence_scores.get(cluster_id, 0)
            applicant_info = self.cluster_applicants.get(cluster_id, {})

            print(f"CLUSTER {cluster_id} (Semantic Coherence:
{coherence:.3f}):")
            print(f"  Size: {len(cluster_data)} patents")
            print(f"  Key concepts: {', '.join([term for term, _ in
keywords[:7]])}")

            # Date analysis
            if 'earliest_claim_date' in cluster_data.columns:
                date_range = f"{cluster_data['earliest_claim_date'].min()}
to {cluster_data['earliest_claim_date'].max()}"
                print(f"  Timeline: {date_range}")

            # Enhanced applicant analysis
            if 'top_applicants' in applicant_info:
                print(f"  Unique applicants:
{applicant_info.get('unique_applicants', 'N/A')}")
                top_applicants = applicant_info['top_applicants']

                print(f"  Leading applicants:")
                for applicant, count in list(top_applicants.items())[:5]:
                    percentage = (count / len(cluster_data)) * 100
```

```
                    print(f"    - {applicant}: {count} patents
({percentage:.1f}%)")

            # Calculate market concentration (Herfindahl index)
            if 'top_applicants' in applicant_info:
                total_patents = len(cluster_data)
                hhi = sum((count/total_patents)**2 for count in
applicant_info['top_applicants'].values())
                concentration_level = "High" if hhi > 0.25 else "Medium" if
hhi > 0.15 else "Low"
                print(f"  Market concentration: {concentration_level} (HHI:
{hhi:.3f})")

            # Top CPC symbols analysis (unchanged)
            if 'cpc_symbols' in cluster_data.columns:
                cpc_symbol_counts = {}
                for cpc_list in cluster_data['cpc_symbols'].dropna():
                    if isinstance(cpc_list, list):
                        unique_symbols = set(cpc_list)
                        for symbol in unique_symbols:
                            cpc_symbol_counts[symbol] =
cpc_symbol_counts.get(symbol, 0) + 1
                    elif isinstance(cpc_list, str):
                        cpc_symbol_counts[cpc_list] =
cpc_symbol_counts.get(cpc_list, 0) + 1

                if cpc_symbol_counts:
                    sorted_cpc = sorted(cpc_symbol_counts.items(),
key=lambda x: x[1], reverse=True)[:3]
                    top_cpc = ', '.join([f"{symbol} ({count})" for symbol,
count in sorted_cpc])
                    print(f"  Top CPC symbols: {top_cpc}")

            # Sample patents
            sample_titles =
cluster_data['invention_title_text'].head(2).tolist()
            print(f"  Example patents:")
            for title in sample_titles:
                print(f"    - {title}")

            print()

    def create_cluster_statistics_plot(self):
        """Create additional statistical plots for cluster analysis"""
        try:
            # Get cluster statistics
            cluster_stats = []
            for cluster_id in sorted(self.data['cluster'].unique()):
                if cluster_id == -1:
```

```
                    continue
                cluster_data = self.data[self.data['cluster'] == cluster_id]
                cluster_stats.append({
                    'cluster_id': cluster_id,
                    'size': len(cluster_data),
                    'coherence':
self.analyze_cluster_semantic_coherence().get(cluster_id, 0)
                })

            if not cluster_stats:
                return

            cluster_df = pd.DataFrame(cluster_stats)

            # Create subplots for statistics
            fig, axes = plt.subplots(2, 2, figsize=(15, 10))

            # 1. Cluster sizes
            axes[0, 0].bar(cluster_df['cluster_id'], cluster_df['size'],
                           color='skyblue', edgecolor='black', alpha=0.7)
            axes[0, 0].set_title('Cluster Sizes', fontsize=12,
fontweight='bold')
            axes[0, 0].set_xlabel('Cluster ID')
            axes[0, 0].set_ylabel('Number of Patents')
            axes[0, 0].grid(True, alpha=0.3)

            # 2. Coherence scores
            axes[0, 1].bar(cluster_df['cluster_id'],
cluster_df['coherence'],
                           color='lightcoral', edgecolor='black', alpha=0.7)
            axes[0, 1].set_title('Cluster Coherence Scores', fontsize=12,
fontweight='bold')
            axes[0, 1].set_xlabel('Cluster ID')
            axes[0, 1].set_ylabel('Coherence Score')
            axes[0, 1].grid(True, alpha=0.3)

            # 3. Size vs Coherence scatter
            axes[1, 0].scatter(cluster_df['size'], cluster_df['coherence'],
                               s=100, alpha=0.7, color='green',
edgecolors='black')
            axes[1, 0].set_title('Cluster Size vs Coherence', fontsize=12,
fontweight='bold')
            axes[1, 0].set_xlabel('Cluster Size')
            axes[1, 0].set_ylabel('Coherence Score')
            axes[1, 0].grid(True, alpha=0.3)

            # Add cluster ID labels to scatter plot
            for _, row in cluster_df.iterrows():
                axes[1, 0].annotate(f"C{int(row['cluster_id'])}",
```

```
                                   (row['size'], row['coherence']),
                                   xytext=(5, 5), textcoords='offset
points',
                                   fontsize=9, fontweight='bold')

            # 4. Distribution histogram
            axes[1, 1].hist(cluster_df['size'], bins=max(5,
len(cluster_df)//3),
                           color='orange', alpha=0.7, edgecolor='black')
            axes[1, 1].set_title('Distribution of Cluster Sizes',
fontsize=12, fontweight='bold')
            axes[1, 1].set_xlabel('Cluster Size')
            axes[1, 1].set_ylabel('Frequency')
            axes[1, 1].grid(True, alpha=0.3)

            plt.suptitle('Cluster Analysis Statistics', fontsize=16,
fontweight='bold')
            plt.tight_layout()
            plt.savefig('cluster_statistics.png', dpi=300,
bbox_inches='tight')
            plt.show()
            print("Cluster statistics visualization: SUCCESS")

        except Exception as e:
            print(f"Failed to create cluster statistics plot: {e}")

# Example usage
if __name__ == "__main__":
    # Initialize semantic clusterer
    clusterer = SemanticPatentClusterer(
        '/content/av_patentdata.jsonl',
        model_name='all-MiniLM-L6-v2'  # Fast and efficient
        # model_name='all-mpnet-base-v2'  # Higher quality, slower
    )

    # Run complete semantic analysis
    results = clusterer.run_complete_semantic_analysis(
        visualization_method='umap'  # or 'tsne'
    )

    print("Semantic analysis completed!")
    print(f"Embedding dimension: {results['embeddings'].shape[1]}")
    print(f"Number of clusters: {len(np.unique(clusterer.cluster_labels))}")
```

Innovation Trend Analysis (NLP):

```
import json
import pandas as pd
import numpy as np
import re
```

```python
from datetime import datetime
from scipy.stats import pearsonr
from scipy.spatial.distance import cosine
from sklearn.feature_extraction.text import TfidfVectorizer
from sentence_transformers import SentenceTransformer
import matplotlib.pyplot as plt
import spacy
from collections import Counter, defaultdict
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.tag import pos_tag
from nltk.chunk import ne_chunk
from nltk.tree import Tree

# Download required NLTK data
try:
    nltk.data.find('tokenizers/punkt')
    nltk.data.find('taggers/averaged_perceptron_tagger')
    nltk.data.find('chunkers/maxent_ne_chunker')
    nltk.data.find('corpora/words')
    nltk.data.find('corpora/stopwords')
except LookupError:
    nltk.download('punkt')
    nltk.download('averaged_perceptron_tagger')
    nltk.download('maxent_ne_chunker')
    nltk.download('words')
    nltk.download('stopwords')


class EnhancedInnovationTrendAnalyzer:
    """
    Enhanced Innovation Trend Analysis focusing on meaningful technical
terms
    Extracts noun phrases, technical concepts, and domain-specific
terminology
    """

    def __init__(self, file_path, time_window='yearly',
min_patents_per_period=5):
        """
        Initialize Enhanced Innovation Trend Analyzer

        Args:
            file_path: Path to JSONL patent data file
            time_window: 'yearly', 'quarterly', or custom period in months
            min_patents_per_period: Minimum patents required per time period
        """
        self.file_path = file_path
```

```
        self.time_window = time_window
        self.min_patents_per_period = min_patents_per_period

        # Core data structures
        self.data = None
        self.processed_texts = None
        self.time_periods = []
        self.period_data = {}

        # NLP components
        self.nlp = None
        self.stop_words = set(stopwords.words('english'))
        self.custom_stopwords = {
            'within', 'wherein', 'thereof', 'therein', 'whereby', 'thereby',
            'herein', 'among', 'wherein', 'upon', 'thereafter', 'therefrom',
            'wherefrom', 'hereby', 'therewith', 'hereafter', 'herewith'
        }
        self.stop_words.update(self.custom_stopwords)


        # Technical term filters
        # Enhanced generic_terms - expanded to catch more patent boilerplate
        self.generic_terms = {
            # Basic patent language
            'method', 'system', 'apparatus', 'device', 'process', 'means',
            'step', 'using', 'determine', 'generate', 'generating',
'provide',
            'include', 'perform', 'obtain', 'receive', 'send', 'configured',
            'based', 'plurality', 'corresponding', 'associated', 'related',
            'first', 'second', 'third', 'one', 'two', 'three', 'embodiment',
            'invention', 'patent', 'claim', 'figure', 'example', 'present',
            'disclosure', 'described', 'accordance', 'aspect', 'particular',

            # Generic verb-noun combinations that appear meaningless
            'provide instruction', 'receive signal', 'determine output',
'process data',
            'generate information', 'obtain data', 'perform operation',
'include step',
            'configure system', 'execute instruction', 'store information',

            # Generic prepositional phrases
            'in accordance with', 'with respect to', 'in order to', 'based
on',
            'corresponding to', 'associated with', 'related to', 'configured
to',

            # Patent boilerplate instruction phrases
            'instructions cause', 'executable instructions', 'program
instructions',
```

```
            'computer instructions', 'medium instructions', 'thereon
instructions',
            'stored instructions', 'software instructions', 'code
instructions',

            # Generic data/output terms
            'output data', 'input data', 'data output', 'data input',
'information data',
            'output information', 'input information', 'data stream', 'data
flow',

            # Generic temporal terms
            'time information', 'time data', 'time duration', 'time period',
'time interval',
            'current time', 'predetermined time', 'specific time',

            # Keep AV-specific generic terms but make them more targeted
            # (removing overly broad terms like 'vehicle', 'driving' that
might filter good compounds)
            'automobile', 'auto', 'transport', 'transportation'
        }

        # Analysis components
        self.meaningful_terms = {}
        self.noun_phrases = {}
        self.technical_concepts = {}
        self.domain_terms = {}

        # Results storage
        self.trending_terms = {}
        self.declining_terms = {}
        self.trending_concepts = {}
        self.declining_concepts = {}
        self.applicant_trends = {}
        self.technology_evolution = {}

        # Initialize NLP pipeline
        self._initialize_nlp()

        print("=== ENHANCED INNOVATION TREND ANALYZER INITIALIZED ===")
        print(f"Time window: {time_window}")
        print(f"Minimum patents per period: {min_patents_per_period}")

    def _initialize_nlp(self):
        """Initialize spaCy NLP pipeline"""
        try:
            # Try to load English model
            self.nlp = spacy.load("en_core_web_sm")
            print("spaCy English model loaded successfully")
```

```
        except OSError:
            print("spaCy English model not found. Please install with:")
            print("python -m spacy download en_core_web_sm")
            print("Falling back to NLTK-based processing...")
            self.nlp = None

    def load_data(self, file_path):
        """Load and parse JSONL file"""
        print("Loading patent data...")
        data = []
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                data.append(json.loads(line.strip()))

        df = pd.DataFrame(data)
        print(f"Loaded {len(df)} patents")
        return df

    def extract_meaningful_terms_spacy(self, text):
      """Extract meaningful terms using spaCy"""
      if not self.nlp or not text:
          return []

      doc = self.nlp(text)
      candidates = set()

      # Extract noun chunks (these preserve proper word order)
      for chunk in doc.noun_chunks:
          phrase = chunk.text.lower().strip()
          if 2 <= len(phrase.split()) <= 4 and len(phrase) > 5:
              if not any(generic in phrase for generic in
self.generic_terms):
                  candidates.add(phrase)

      # Extract compound terms - FIXED: preserve head-child relationship
order
      for token in doc:
          if token.pos_ == 'NOUN' and token.head.pos_ in {'ADJ', 'NOUN'}:
              # Check dependency relation to preserve correct order
              if token.dep_ in {'compound', 'amod'}:  # token modifies head
                  head = token.head.text.lower()
                  child = token.text.lower()
                  compound = f"{child} {head}"  # modifier comes first
              else:  # head modifies token
                  head = token.head.text.lower()
                  child = token.text.lower()
                  compound = f"{head} {child}"  # head comes first

              if len(compound.split()) == 2 and len(compound) > 6:
```

```
                  if not any(generic in compound for generic in
self.generic_terms):
                      candidates.add(compound)

      # Named entities: Only include useful short entities
      for ent in doc.ents:
          if ent.label_ in {'ORG', 'PRODUCT', 'TECH'} and
len(ent.text.split()) <= 3:
              ent_text = ent.text.lower().strip()
              if not any(generic in ent_text for generic in
self.generic_terms):
                  candidates.add(ent_text)

      return list(candidates)

    def extract_meaningful_terms_nltk(self, text):
        """Extract meaningful terms using NLTK (fallback)"""
        if not text:
            return []

        tokens = word_tokenize(text.lower())
        pos_tags = pos_tag(tokens)
        candidates = set()

        # Extract noun phrase sequences (e.g. [JJ]* [NN]+)
        current_phrase = []
        for word, pos in pos_tags:
            if pos.startswith('JJ') or pos.startswith('NN'):
                current_phrase.append(word)
            else:
                if 2 <= len(current_phrase) <= 4:
                    phrase = ' '.join(current_phrase)
                    if len(phrase) > 5 and not any(generic in phrase for
generic in self.generic_terms):
                        candidates.add(phrase)
                current_phrase = []
        # Final phrase at the end
        if 2 <= len(current_phrase) <= 4:
            phrase = ' '.join(current_phrase)
            if len(phrase) > 5 and not any(generic in phrase for generic in
self.generic_terms):
                candidates.add(phrase)

        # Extract technical compound patterns
        for i in range(len(pos_tags) - 1):
            word1, pos1 = pos_tags[i]
            word2, pos2 = pos_tags[i + 1]
            if ((pos1.startswith('NN') and pos2.startswith('NN')) or
                (pos1.startswith('JJ') and pos2.startswith('NN')) or
```

```
                (pos1.startswith('NN') and pos2 == 'VBG')):

                compound = f"{word1} {word2}"
                if len(compound) > 6 and not any(generic in compound for
generic in self.generic_terms):
                    candidates.add(compound)

        return list(candidates)

    def extract_domain_specific_terms(self, text):
        """Extract domain-specific technical terms"""
        domain_patterns = {
            # Perception & Sensing - expanded with variations
            'perception': r'\b(lidar|li-dar|light detection
ranging|radar|radio detection|'
                        r'camera system|vision system|stereo camera|mono
camera|'
                        r'ultrasonic sensor|sonar|stereo vision|binocular
vision|'
                        r'depth estimation|depth perception|distance
measurement|'
                        r'object detection|target detection|obstacle
detection|'
                        r'lane detection|lane recognition|road marking
detection|'
                        r'traffic sign recognition|sign detection|signal
recognition|'
                        r'pedestrian detection|person detection|human
detection|'
                        r'cyclist detection|bicycle detection|bike
detection|'
                        r'semantic segmentation|pixel classification|scene
parsing|'
                        r'instance segmentation|object segmentation|'
                        r'point cloud|3d point cloud|laser scan|lidar scan|'
                        r'sensor fusion|multi sensor|data fusion|information
fusion|'
                        r'multi-modal sensing|multimodal perception|'
                        r'occupancy grid|occupancy map|grid map|voxel
grid|3d grid)\b',

            # Localization & Mapping - with abbreviations
            'localization': r'\b(simultaneous localization
mapping|slam|visual slam|v-slam|'
                          r'lidar slam|l-slam|visual odometry|vo|stereo
odometry|'
                          r'gps rtk|rtk gps|real time kinematic|differential
gps|dgps|'
```

```
                          r'high definition map|hd map|high precision
map|detailed map|'
                          r'localization accuracy|pose accuracy|position
accuracy|'
                          r'map matching|route matching|path matching|'
                          r'dead reckoning|inertial navigation|ins|'
                          r'particle filter|monte carlo|kalman
filter|extended kalman|'
                          r'loop closure|place recognition|relocalization|'
                          r'pose estimation|position estimation|orientation
estimation|'
                          r'global localization|local localization)\b',

            # Path Planning & Control - expanded control terms
            'planning_control': r'\b(path planning|route planning|trajectory
planning|'
                              r'motion planning|behavioral
planning|strategic planning|'
                              r'route optimization|path
optimization|trajectory optimization|'
                              r'decision making|behavior planning|maneuver
planning|'
                              r'steering control|lateral
control|longitudinal control|'
                              r'throttle control|acceleration control|speed
control|'
                              r'brake control|braking control|deceleration
control|'
                              r'pid controller|proportional integral|model
predictive control|'
                              r'mpc|linear quadratic|lqr|optimal control|'
                              r'collision avoidance|obstacle avoidance|crash
avoidance|'
                              r'emergency braking|automatic emergency|aeb|'
                              r'lane keeping|lane centering|lane following|'
                              r'adaptive cruise control|acc|cruise control|'
                              r'trajectory tracking|path following|reference
tracking)\b',

            # AI/ML for AV - updated with latest architectures
            'av_ai': r'\b(deep neural network|dnn|convolutional
neural|cnn|convnet|'
                    r'recurrent neural|rnn|lstm|gru|transformer model|'
                    r'vision transformer|vit|attention mechanism|self
attention|'
                    r'reinforcement learning|rl|deep reinforcement|drl|'
                    r'imitation learning|behavioral cloning|inverse
reinforcement|'
                    r'end to end learning|e2e learning|end-to-end|'
```

```
                  r'supervised learning|unsupervised learning|semi
supervised|'
                  r'transfer learning|domain adaptation|fine tuning|'
                  r'adversarial training|generative adversarial|gan|'
                  r'generative model|variational autoencoder|vae|'
                  r'graph neural network|gnn|graph convolution|'
                  r'temporal modeling|sequence modeling|time series|'
                  r'sequence prediction|future prediction|motion
prediction)\b',

            # V2X Communication - expanded protocols
            'v2x': r'\b(vehicle to vehicle|v2v|car to car|c2c|'
                  r'vehicle to infrastructure|v2i|vehicle to roadside|v2r|'
                  r'vehicle to everything|v2x|vehicle to cloud|v2c|'
                  r'vehicle to pedestrian|v2p|vehicle to network|v2n|'
                  r'dedicated short range|dsrc|wave|ieee 802.11p|'
                  r'cellular v2x|c-v2x|lte v2x|5g v2x|nr v2x|'
                  r'5g communication|lte communication|cellular
communication|'
                  r'cooperative driving|coordinated driving|collaborative
driving|'
                  r'platooning|convoy|vehicle following|'
                  r'vehicle coordination|traffic coordination|fleet
coordination|'
                  r'intersection management|traffic light|signal phase|'
                  r'connected vehicle|connected car|iot vehicle|'
                  r'communication protocol|message protocol|data
exchange)\b',

            # Safety & Validation - expanded standards
            'safety': r'\b(functional safety|safety function|iso 26262|iec
61508|'
                    r'safety integrity level|sil|automotive safety
integrity|asil|'
                    r'hazard analysis|hazop|fault tree|fmea|failure mode|'
                    r'risk assessment|safety assessment|risk analysis|'
                    r'fault tolerance|fault tolerant|error tolerance|'
                    r'redundancy|backup system|failover|'
                    r'fail safe|fail operational|fail silent|graceful
degradation|'
                    r'safety monitoring|health monitoring|diagnostic
monitoring|'
                    r'verification validation|v&v|testing validation|'
                    r'scenario testing|test scenario|corner case|edge case|'
                    r'safety critical|mission critical|life critical|'
                    r'automotive safety|vehicle safety|driving safety)\b',

            # Simulation & Testing - expanded platforms
```

```
            'simulation': r'\b(virtual environment|simulation
environment|test environment|'
                        r'simulation platform|testing
platform|carla|airsim|sumo|'
                        r'digital twin|virtual twin|cyber physical|'
                        r'synthetic data generation|artificial
data|simulated data|'
                        r'procedural generation|automatic generation|'
                        r'physics simulation|dynamics simulation|vehicle
dynamics|'
                        r'sensor simulation|lidar simulation|camera
simulation|'
                        r'traffic simulation|behavior simulation|scenario
simulation|'
                        r'scenario generation|test case generation|'
                        r'test automation|automated testing|continuous
testing|'
                        r'hardware in loop|hil|software in loop|sil|'
                        r'closed loop testing|open loop testing|'
                        r'model in loop|mil|processor in loop|pil)\b',

            # Cybersecurity for AVs - new domain
            'cybersecurity': r'\b(automotive cybersecurity|vehicle
security|can security|'
                            r'intrusion detection|anomaly detection|security
monitoring|'
                            r'encryption|authentication|authorization|pki|'
                            r'secure communication|secure boot|trusted
execution|'
                            r'firewall|security gateway|security
module|hsm|'
                            r'penetration testing|vulnerability assessment|'
                            r'security update|over air|ota security)\b',

            # Human-Machine Interface - new domain
            'hmi': r'\b(human machine interface|hmi|driver monitoring|driver
attention|'
                  r'takeover request|handover|mode transition|'
                  r'user experience|ux|human factors|ergonomics|'
                  r'driver assistance|adas|level 2|level 3|level 4|level 5|'
                  r'situational awareness|trust|acceptance|usability)\b'
        }

        domain_terms = []
        text_lower = text.lower()

        for domain, pattern in domain_patterns.items():
            matches = re.finditer(pattern, text_lower)
            for match in matches:
```

```
                term = match.group().strip()
                if len(term) > 3 and self.is_technically_relevant(term) and
self._has_proper_phrase_structure(term.split()):
                    domain_terms.append(term)


        return list(set(domain_terms))

    def preprocess_patent_text(self, text):
        """Enhanced preprocessing for meaningful term extraction"""
        if pd.isna(text) or text == '':
            return ''

        # Normalize technical references but keep them meaningful
        text = re.sub(r'\b(fig\.|figure)\s*\d+\b', 'figure', text,
flags=re.IGNORECASE)
        text = re.sub(r'\bclaim\s*\d+\b', 'patent_claim', text,
flags=re.IGNORECASE)
        text = re.sub(r'\bpatent\s*\d+\b', 'patent_reference', text,
flags=re.IGNORECASE)

        # Preserve technical terms with numbers
        text = re.sub(r'\b([a-zA-Z_]+)\s+(\d+)\b', r'\1_\2', text)

        # Clean but preserve hyphens in technical terms
        text = re.sub(r'[^\w\s\-\.]', ' ', text)
        text = re.sub(r'\s+', ' ', text).strip()

        return text.lower()

    def extract_all_meaningful_terms(self, text):
        """Extract all types of meaningful terms from text"""
        all_terms = []

        # Method 1: spaCy-based extraction (if available)
        if self.nlp:
            spacy_terms = self.extract_meaningful_terms_spacy(text)
            all_terms.extend(spacy_terms)
        else:
            # Method 2: NLTK-based extraction (fallback)
            nltk_terms = self.extract_meaningful_terms_nltk(text)
            all_terms.extend(nltk_terms)

        # Method 3: Domain-specific pattern matching
        domain_terms = self.extract_domain_specific_terms(text)
        all_terms.extend(domain_terms)

        # Method 4: Technical n-grams (2-3 grams with technical relevance)
        technical_ngrams = self.extract_technical_ngrams(text)
```

```
        all_terms.extend(technical_ngrams)

        # Remove duplicates and filter
        unique_terms = list(set(all_terms))
        filtered_terms = self.filter_meaningful_terms(unique_terms)

        return filtered_terms

    def extract_technical_ngrams(self, text, n_range=(2, 3)):
        """Extract technically relevant n-grams using POS tags (e.g.,
Adj+Noun, Noun+Noun)"""
        technical_indicators = {
            # Core AV Systems
            'perception system', 'sensing system', 'vision system', 'lidar
system',
            'planning system', 'control system', 'localization system',
'navigation system',
            'sensor array', 'sensor suite', 'sensor cluster', 'multi
sensor',
            'fusion algorithm', 'detection algorithm', 'tracking algorithm',
            'prediction algorithm', 'decision algorithm', 'planning
algorithm',

            # Advanced Data Processing
            'point cloud', 'feature extraction', 'feature detection',
'feature matching',
            'object classification', 'scene understanding', 'scene
analysis',
            'temporal consistency', 'spatial reasoning', 'geometric
reasoning',
            'multi frame', 'sequential data', 'time series', 'temporal
modeling',
            'real time processing', 'edge computing', 'onboard computing',
'embedded computing',

            # Vehicle Components & Systems
            'electronic control unit', 'ecu', 'domain controller', 'central
computer',
            'actuator control', 'steering actuator', 'brake actuator',
'throttle actuator',
            'vehicle dynamics', 'chassis control', 'powertrain control',
'drivetrain control',
            'thermal management', 'power management', 'battery management',

            # Communication & Connectivity
            'wireless communication', 'cellular communication', 'v2x
communication',
            'network protocol', 'communication protocol', 'data
transmission',
```

```
            'signal processing', 'antenna system', 'telematics unit',
'connectivity module',
            'over air update', 'remote update', 'software update',

            # Safety & Monitoring Systems
            'safety monitor', 'diagnostic system', 'health monitoring',
'condition monitoring',
            'fault detection', 'error detection', 'anomaly detection',
'intrusion detection',
            'system validation', 'performance monitoring', 'reliability
monitoring',

            # Infrastructure & Environment
            'road infrastructure', 'smart infrastructure', 'intelligent
infrastructure',
            'traffic management', 'intersection control', 'signal control',
            'parking system', 'charging infrastructure', 'energy
management',

            # AI/ML Specific Systems
            'neural network', 'deep learning', 'machine learning', 'computer
vision',
            'pattern recognition', 'image processing', 'data mining',
'knowledge extraction',
            'model training', 'model inference', 'model deployment', 'model
optimization',

            # Testing & Validation
            'simulation platform', 'testing framework', 'validation
framework',
            'scenario testing', 'hardware testing', 'software testing',
'system testing'
        }

        technical_ngrams = []

        if not self.nlp:
            return technical_ngrams  # fallback not supported in this mode

        doc = self.nlp(text)

        for i in range(len(doc)):
            for n in range(n_range[0], n_range[1] + 1):
                if i + n > len(doc):
                    continue

                span = doc[i:i+n]
                span_text = span.text.strip()
```

```
                # Skip if too short or contains digits
                if len(span_text) <= 8 or any(char.isdigit() for char in
span_text):
                    continue

                # POS pattern check: allow (Adj|Noun)+ Noun patterns
                pos_tags = [token.pos_ for token in span]
                if not pos_tags[-1].startswith('NOUN'):
                    continue  # final word must be a noun
                if not all(pos in {'ADJ', 'NOUN'} for pos in pos_tags):
                    continue  # only adj/noun sequences allowed

                # Stopword & generic filters
                if any(token.lower_ in self.stop_words for token in span):
                    continue
                if any(generic in span_text.lower() for generic in
self.generic_terms):
                    continue

                # Must contain a technical indicator
                if any(indicator in span_text.lower() for indicator in
technical_indicators):
                    technical_ngrams.append(span_text)

        return technical_ngrams

    def filter_meaningful_terms(self, terms):
        """Filter terms to keep only meaningful ones and avoid flipped or
repetitive junk"""
        filtered = []
        seen_terms = set()
        word_overlap_groups = {}

        # First pass: group terms by significant word overlap
        for term in terms:
            term = term.strip()

            # Length constraints
            if len(term) < 5 or len(term) > 50:
                continue

            # Stop if mostly generic
            if any(generic in term.lower() for generic in
self.generic_terms):
                continue

            words = term.split()
            lower_words = [w.lower() for w in words]
```

```python
            # Must be at least bi-gram
            if len(words) < 2:
                continue

            # Remove leading/trailing stopwords
            if lower_words[0] in self.stop_words or lower_words[-1] in
self.stop_words:
                continue

            # Avoid phrases with unnecessary articles
            if any(w in {'the', 'a', 'an'} for w in lower_words):
                continue

            # Skip if all words are the same or repeated
            if len(set(lower_words)) == 1:
                continue
            if any(lower_words[i] == lower_words[i + 1] for i in
range(len(lower_words) - 1)):
                continue

            # Skip exact duplicates
            term_lower = term.lower()
            if term_lower in seen_terms:
                continue
            seen_terms.add(term_lower)

            # Final technical relevance check
            if not self.is_technically_relevant(term):
                continue

            # Group by word overlap for deduplication
            word_set = set(lower_words)
            found_group = False

            for existing_group_key, existing_terms in
word_overlap_groups.items():
                existing_word_set = set(existing_group_key.split())

                # Check for significant overlap (share >= 50% of words)
                overlap = len(word_set.intersection(existing_word_set))
                min_words = min(len(word_set), len(existing_word_set))

                if overlap >= max(1, min_words * 0.5):  # At least 50%
overlap or 1 word minimum
                    existing_terms.append((term, len(words), word_set))
                    found_group = True
                    break

            if not found_group:
```

```
                word_overlap_groups[' '.join(sorted(word_set))] = [(term,
len(words), word_set)]

        # Second pass: select best term from each group
        for group_terms in word_overlap_groups.values():
            if len(group_terms) == 1:
                filtered.append(group_terms[0][0])
            else:
                # Multiple terms with overlapping words - choose the best
one
                best_term = self._select_best_overlapping_term(group_terms)
                if best_term:
                    filtered.append(best_term)

        return filtered

    def _select_best_overlapping_term(self, overlapping_terms):
        """Select the best term from a group of overlapping terms"""
        # Scoring criteria (higher is better):
        # 1. More specific/longer phrases preferred
        # 2. Proper word order (noun phrases should end with noun)
        # 3. Avoid reversed/awkward phrases

        scored_terms = []

        for term, word_count, word_set in overlapping_terms:
            score = 0
            words = term.lower().split()

            # Length bonus (longer phrases are often more specific)
            score += word_count * 2

            # Proper phrase structure bonus
            if self._has_proper_phrase_structure(words):
                score += 5

            # Technical term positioning bonus
            technical_words = {
                'computing', 'processing', 'learning', 'detection',
'control', 'system',
                'network', 'sensing', 'perception', 'planning',
'navigation', 'tracking',
                'fusion', 'optimization', 'prediction', 'classification',
'recognition',
                'monitoring', 'management', 'coordination', 'communication',
'simulation'
            }
            if any(tech_word in words for tech_word in technical_words):
```

```
                # Bonus if technical words are in proper position (usually
at the end)
                for i, word in enumerate(words):
                    if word in technical_words and i == len(words) - 1:
                        score += 3
                    elif word in technical_words:
                        score += 1

            # Penalize awkward word orders
            if self._is_awkward_phrase(words):
                score -= 3

            scored_terms.append((term, score))

        # Return the highest scoring term
        best_term = max(scored_terms, key=lambda x: x[1])
        return best_term[0]

    def _has_proper_phrase_structure(self, words):
        """Check if phrase has proper grammatical structure"""
        if not self.nlp:
            # Simple heuristic: common technical phrase patterns
            tech_endings = {
                'system', 'network', 'algorithm', 'method', 'process',
'control',
                'detection', 'learning', 'computing', 'processing',
'sensing', 'perception',
                'planning', 'navigation', 'tracking', 'fusion',
'optimization', 'prediction',
                'classification', 'recognition', 'monitoring', 'management',
'coordination',
                'communication', 'simulation', 'platform', 'framework',
'module', 'unit',
                'interface', 'protocol', 'function', 'service',
'application', 'solution'
            }
            return words[-1] in tech_endings

        # Use spaCy for more accurate analysis
        doc = self.nlp(' '.join(words))
        pos_tags = [token.pos_ for token in doc]

        # Good patterns: ADJ* NOUN+, NOUN+ NOUN, etc.
        if len(pos_tags) >= 2:
            # Should generally end with a noun
            if pos_tags[-1].startswith('NOUN'):
                return True
            # Or verb form used as noun (like "learning", "processing")
```

```
            if pos_tags[-1] in ['VBG'] and words[-1] in {'learning',
'processing', 'computing', 'training', 'sensing', 'tracking',
                                                        'planning',
'monitoring', 'mapping', 'routing', 'steering', 'braking',
                                                        'charging',
'parking', 'following', 'avoiding', 'detecting', 'recognizing',
                                                        'classifying',
'segmenting', 'filtering', 'optimizing', 'predicting'}:
                return True

        return False

    def _is_awkward_phrase(self, words):
        """Detect awkward or reversed phrases"""
        awkward_patterns = [
            # Resource/computing reversals
            (lambda w: 'resources' in w and 'cloud' in w and
w.index('resources') < w.index('cloud')),
            (lambda w: 'operations' in w and any(tech in w for tech in
['learning', 'computing']) and
            'operations' in w[:len(w)//2]),  # operations at the beginning
is often awkward
            # Add more patterns as needed
        ]

        for pattern in awkward_patterns:
            if pattern(words):
                return True

        return False

    def is_technically_relevant(self, term):
        """Check if a term is technically relevant"""
        technical_keywords = {
            # Sensors & Hardware
            'lidar', 'radar', 'camera', 'ultrasonic', 'sonar', 'imu',
'gnss', 'gps',
            'accelerometer', 'gyroscope', 'magnetometer', 'odometer',
'encoder',
            'processor', 'gpu', 'tpu', 'fpga', 'asic', 'soc', 'embedded',
'microcontroller',
            'ecu', 'domain controller', 'central computer', 'edge computer',
            'actuator', 'servo', 'motor', 'brake', 'steering', 'throttle',
'suspension',

            # Software & Algorithms
            'algorithm', 'neural', 'network', 'learning', 'training',
'inference',
```

```
            'optimization', 'calibration', 'fusion', 'filtering',
'estimation',
            'prediction', 'classification', 'segmentation', 'tracking',
'detection',
            'recognition', 'matching', 'clustering', 'regression',
'clustering',

            # AV-Specific Technical Concepts
            'autonomous', 'automated', 'self driving', 'driverless',
'unmanned',
            'adas', 'level 2', 'level 3', 'level 4', 'level 5', 'sae
levels',
            'trajectory', 'waypoint', 'route', 'navigation', 'guidance',
'localization',
            'slam', 'odometry', 'mapping', 'path planning', 'motion
planning',
            'collision', 'obstacle', 'hazard', 'safety', 'emergency',
'takeover',
            'lane', 'intersection', 'parking', 'overtaking', 'merging',
'platooning',

            # Perception & Understanding
            'point cloud', 'voxel', 'occupancy grid', 'semantic',
'instance',
            'object detection', 'lane detection', 'sign recognition', 'depth
estimation',
            'scene understanding', 'situational awareness', 'environment
modeling',

            # Data & Processing
            'sensor data', 'telemetry', 'logging', 'recording',
'annotation', 'labeling',
            'dataset', 'training data', 'validation', 'testing',
'simulation',
            'modeling', 'visualization', 'analysis', 'processing',
'computation',
            'real time', 'latency', 'throughput', 'bandwidth', 'memory',
'storage',

            # Communication & Connectivity
            'wireless', 'cellular', 'wifi', 'bluetooth', 'ethernet', 'can
bus', 'flexray',
            'lin', 'most', 'automotive ethernet', 'protocol', 'interface',
'gateway',
            'cloud', 'edge', 'fog', 'v2x', 'v2v', 'v2i', 'dsrc', 'c-v2x',

            # Safety & Standards
            'iso', 'sae', 'nhtsa', 'dot', 'ece', 'regulation', 'standard',
'compliance',
```

```
            'certification', 'homologation', 'type approval', 'functional
safety',
            'iso 26262', 'asil', 'sil', 'hazop', 'fmea', 'redundancy',
'fault tolerance',

            # Testing & Validation
            'simulation', 'virtual', 'synthetic', 'scenario', 'test case',
'corner case',
            'edge case', 'verification', 'validation', 'hil', 'sil', 'mil',
'pil',

            # Human Factors
            'hmi', 'driver monitoring', 'attention', 'drowsiness',
'distraction',
            'takeover', 'handover', 'trust', 'acceptance', 'usability',
'ergonomics',

            # Cybersecurity
            'cybersecurity', 'security', 'encryption', 'authentication',
'authorization',
            'firewall', 'intrusion', 'vulnerability', 'penetration', 'secure
boot',

            # Energy & Sustainability
            'electric', 'hybrid', 'battery', 'charging', 'energy
management',
            'regenerative braking', 'efficiency', 'range', 'consumption'
        }

        # Check if term contains technical keywords
        term_lower = term.lower()
        has_technical = any(keyword in term_lower for keyword in
technical_keywords)

        # Check for technical patterns (e.g., contains numbers, technical
suffixes)
        has_technical_pattern = (
            re.search(r'\d', term) or  # Contains numbers
            term.endswith(('_based', '_enabled', '_driven', '_aware')) or
            any(suffix in term for suffix in ['tion', 'ing', 'ment', 'ness',
'ity'])
        )

        return has_technical or has_technical_pattern

    def prepare_temporal_corpus(self):
        """Prepare text corpus with enhanced term extraction"""
        print("Preparing temporal corpus with meaningful term
extraction...")
```

```
        # Load and preprocess data
        self.data = self.load_data(self.file_path)

        # Convert dates
        self.data['date_published'] =
pd.to_datetime(self.data['date_published'])
        self.data['earliest_claim_date'] =
pd.to_datetime(self.data['earliest_claim_date'])

        # Use earliest claim date as primary temporal marker
        self.data['analysis_date'] =
self.data['earliest_claim_date'].fillna(self.data['date_published'])

        # Combine text fields for analysis
        self.data['claims_text'] = self.data['claims'].apply(
            lambda x: ' '.join(x) if isinstance(x, list) else str(x)
        )

        # Create comprehensive text
        self.data['full_text'] = (
            self.data['invention_title_text'].fillna('') + '. ' +
            self.data['abstract_text'].fillna('') + '. ' +
            self.data['claims_text'].fillna('')
        )

        # Preprocess texts
        self.data['processed_text'] =
self.data['full_text'].apply(self.preprocess_patent_text)

        # Extract meaningful terms for each patent
        print("Extracting meaningful terms from patents...")
        self.data['meaningful_terms'] = self.data['processed_text'].apply(
            self.extract_all_meaningful_terms
        )

        # Filter valid documents
        valid_mask = (
            (self.data['processed_text'].str.len() > 50) &
            (self.data['analysis_date'].notna()) &
            (self.data['meaningful_terms'].apply(len) > 0)
        )
        self.data = self.data[valid_mask].reset_index(drop=True)

        print(f"Prepared {len(self.data)} valid patents for temporal
analysis")

        # Create time periods
        self._create_time_periods()
```

```
        return self.data

    def _create_time_periods(self):
        """Create time period segmentation"""
        print("Creating time period segmentation...")

        min_date = self.data['analysis_date'].min()
        max_date = self.data['analysis_date'].max()

        if self.time_window == 'yearly':
            # Create yearly periods
            start_year = min_date.year
            end_year = max_date.year

            for year in range(start_year, end_year + 1):
                period_start = datetime(year, 1, 1)
                period_end = datetime(year, 12, 31)

                period_data = self.data[
                    (self.data['analysis_date'] >= period_start) &
                    (self.data['analysis_date'] <= period_end)
                ]

                if len(period_data) >= self.min_patents_per_period:
                    period_key = f"{year}"
                    self.time_periods.append(period_key)
                    self.period_data[period_key] = period_data.copy()

        elif self.time_window == 'quarterly':
            # Create quarterly periods
            current_date = min_date.replace(day=1)

            while current_date <= max_date:
                # Determine quarter
                quarter = (current_date.month - 1) // 3 + 1
                quarter_start = datetime(current_date.year, (quarter-1)*3 +
1, 1)

                if quarter == 4:
                    quarter_end = datetime(current_date.year, 12, 31)
                else:
                    quarter_end = datetime(current_date.year, quarter*3, 31)

                period_data = self.data[
                    (self.data['analysis_date'] >= quarter_start) &
                    (self.data['analysis_date'] <= quarter_end)
                ]
```

```
                if len(period_data) >= self.min_patents_per_period:
                    period_key = f"{current_date.year}Q{quarter}"
                    self.time_periods.append(period_key)
                    self.period_data[period_key] = period_data.copy()

                # Move to next quarter
                if quarter == 4:
                    current_date = datetime(current_date.year + 1, 1, 1)
                else:
                    current_date = datetime(current_date.year, quarter*3 +
1, 1)

        self.time_periods.sort()
        print(f"Created {len(self.time_periods)} time periods:
{self.time_periods}")

    def compute_meaningful_term_trends(self):
        """Compute trends for meaningful terms"""
        print("Computing meaningful term trends...")

        # Collect all meaningful terms across all periods
        all_terms = set()
        term_frequencies = defaultdict(lambda: defaultdict(int))

        for period in self.time_periods:
            period_data = self.period_data[period]
            period_terms = []

            # Collect all terms from this period
            for terms_list in period_data['meaningful_terms']:
                period_terms.extend(terms_list)
                all_terms.update(terms_list)

            # Count term frequencies in this period
            term_counts = Counter(period_terms)

            # Normalize by number of patents in period
            num_patents = len(period_data)
            for term, count in term_counts.items():
                term_frequencies[term][period] = count / num_patents

        # Calculate trend scores
        trend_scores = {}

        for term in all_terms:
            # Get scores for each period (0 if term doesn't appear)
            scores = [term_frequencies[term].get(period, 0) for period in
self.time_periods]
```

```
            # Only analyze terms that appear in multiple periods
            non_zero_periods = sum(1 for score in scores if score > 0)
            if non_zero_periods >= 2 and sum(scores) > 0.01:  # Minimum
frequency threshold

                time_indices = list(range(len(scores)))

                if len(set(scores)) > 1:  # Avoid correlation with constant
values
                    try:
                        correlation, p_value = pearsonr(time_indices,
scores)
                        trend_scores[term] = {
                            'correlation': correlation,
                            'p_value': p_value,
                            'mean_score': np.mean(scores),
                            'std_score': np.std(scores),
                            'scores': scores,
                            'non_zero_periods': non_zero_periods,
                            'total_frequency': sum(scores)
                        }
                    except:
                        continue

        # Identify trending terms (positive correlation, significant)
        trending = {
            term: data for term, data in trend_scores.items()
            if (data['correlation'] > 0.4 and
                data['p_value'] < 0.1 and
                data['total_frequency'] > 0.05 and
                data['non_zero_periods'] >= 3)
        }

        # Identify declining terms (negative correlation, significant)
        declining = {
            term: data for term, data in trend_scores.items()
            if (data['correlation'] < -0.4 and
                data['p_value'] < 0.1 and
                data['total_frequency'] > 0.05 and
                data['non_zero_periods'] >= 3)
        }

        # Sort by correlation strength
        self.trending_terms = dict(sorted(trending.items(),
                                        key=lambda x: x[1]['correlation'],
reverse=True))
        self.declining_terms = dict(sorted(declining.items(),
                                         key=lambda x: x[1]['correlation']))
```

```
        print(f"Identified {len(self.trending_terms)} trending meaningful
terms")
        print(f"Identified {len(self.declining_terms)} declining meaningful
terms")

        return trend_scores

    def compute_semantic_trends(self, model_name='all-MiniLM-L6-v2'):
        """Compute semantic embedding based trends using meaningful terms"""
        print("Computing semantic embedding trends...")

        # Initialize SBERT model
        sbert_model = SentenceTransformer(model_name)

        # Compute embeddings for each time period using meaningful terms
        embeddings_by_period = {}

        for period in self.time_periods:
            print(f"Processing period: {period}")

            period_data = self.period_data[period]

            # Create documents from meaningful terms
            period_documents = []
            for terms_list in period_data['meaningful_terms']:
                if terms_list:  # Only non-empty term lists
                    # Join meaningful terms as a document
                    doc = '. '.join(terms_list)
                    period_documents.append(doc)

            if period_documents:
                # Generate embeddings
                embeddings = sbert_model.encode(period_documents,
show_progress_bar=False)
                embeddings_by_period[period] = {
                    'embeddings': embeddings,
                    'mean_embedding': np.mean(embeddings, axis=0),
                    'patents': len(period_documents)
                }

        self.embeddings_by_period = embeddings_by_period

        # Analyze semantic evolution
        self._analyze_semantic_evolution()

        print("Semantic trend analysis completed!")

    def _analyze_semantic_evolution(self):
        """Analyze semantic evolution across time periods"""
```

```
        print("Analyzing semantic evolution...")

        # Calculate period-to-period semantic similarity
        period_similarities = []
        periods = list(self.embeddings_by_period.keys())

        for i in range(len(periods) - 1):
            current_period = periods[i]
            next_period = periods[i + 1]

            current_embedding =
self.embeddings_by_period[current_period]['mean_embedding']
            next_embedding =
self.embeddings_by_period[next_period]['mean_embedding']

            similarity = 1 - cosine(current_embedding, next_embedding)
            period_similarities.append({
                'from_period': current_period,
                'to_period': next_period,
                'similarity': similarity
            })

        self.technology_evolution['period_similarities'] =
period_similarities

        # Identify periods with significant semantic shifts
        if period_similarities:
            similarities = [item['similarity'] for item in
period_similarities]
            mean_similarity = np.mean(similarities)
            std_similarity = np.std(similarities)

            significant_shifts = [
                item for item in period_similarities
                if item['similarity'] < (mean_similarity - std_similarity)
            ]

            self.technology_evolution['significant_shifts'] =
significant_shifts

            print(f"Identified {len(significant_shifts)} periods with
significant semantic shifts")

    def analyze_applicant_trends(self, top_n_applicants=10):
        """Analyze meaningful term trends per applicant"""
        print("Analyzing applicant-specific trends...")

        # Get top applicants by patent count
        applicant_counts = self.data['applicant_name'].value_counts()
```

```
        top_applicants =
applicant_counts.head(top_n_applicants).index.tolist()

        for applicant in top_applicants:
            applicant_data = self.data[self.data['applicant_name'] ==
applicant]

            # Analyze temporal distribution
            applicant_by_period = {}
            for period in self.time_periods:
                period_patents = applicant_data[
                    applicant_data.index.isin(self.period_data[period].index
)
                ]

                if len(period_patents) > 0:
                    # Collect meaningful terms for this applicant in this
period
                    period_terms = []
                    for terms_list in period_patents['meaningful_terms']:
                        period_terms.extend(terms_list)

                    if period_terms:
                        # Get top meaningful terms
                        term_counts = Counter(period_terms)
                        top_terms = term_counts.most_common(10)

                        applicant_by_period[period] = {
                            'patent_count': len(period_patents),
                            'top_meaningful_terms': top_terms,
                            'focus_areas': [term for term, count in
top_terms[:5]]
                        }

            self.applicant_trends[applicant] = applicant_by_period

        print(f"Analyzed trends for {len(top_applicants)} top applicants")

    def generate_trend_visualizations(self):
        """Generate comprehensive trend visualizations"""
        print("Generating trend visualizations...")

        # 1. Trending meaningful terms
        self._plot_trending_meaningful_terms()

        # 2. Declining meaningful terms
        self._plot_declining_meaningful_terms()

        # 3. Semantic evolution visualization
```

```
        self._plot_semantic_evolution()

        # 4. Applicant trend analysis
        self._plot_applicant_trends()

        # 5. Technology shift timeline
        self._plot_technology_shifts()

        print("All visualizations generated!")

    def _plot_trending_meaningful_terms(self, top_n=10):
        """Plot trending meaningful terms over time"""
        plt.figure(figsize=(14, 10))

        # Get top trending terms
        top_trending = list(self.trending_terms.keys())[:top_n]

        for i, term in enumerate(top_trending):
            scores = self.trending_terms[term]['scores']
            correlation = self.trending_terms[term]['correlation']

            plt.plot(self.time_periods, scores,
                    marker='o', linewidth=2,
                    label=f"{term} (r={correlation:.3f})",
                    color=plt.cm.tab10(i))

        plt.title("Top Trending Meaningful Terms Over Time", fontsize=16,
fontweight='bold')
        plt.xlabel("Time Period", fontsize=12)
        plt.ylabel("Normalized Frequency", fontsize=12)
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        plt.xticks(rotation=45)
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()

    def _plot_declining_meaningful_terms(self, top_n=10):
        """Plot declining meaningful terms over time"""
        plt.figure(figsize=(14, 10))

        # Get top declining terms
        top_declining = list(self.declining_terms.keys())[:top_n]

        for i, term in enumerate(top_declining):
            scores = self.declining_terms[term]['scores']
            correlation = self.declining_terms[term]['correlation']

            plt.plot(self.time_periods, scores,
                    marker='o', linewidth=2, linestyle='--',
```

```
                        label=f"{term} (r={correlation:.3f})",
                        color=plt.cm.tab10(i))

        plt.title("Top Declining Meaningful Terms Over Time", fontsize=16,
fontweight='bold')
        plt.xlabel("Time Period", fontsize=12)
        plt.ylabel("Normalized Frequency", fontsize=12)
        plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
        plt.xticks(rotation=45)
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()

    def _plot_semantic_evolution(self):
        """Plot semantic evolution between periods"""
        if 'period_similarities' not in self.technology_evolution:
            return

        similarities = self.technology_evolution['period_similarities']

        periods = [item['from_period'] + ' → ' + item['to_period'] for item
in similarities]
        sim_scores = [item['similarity'] for item in similarities]

        plt.figure(figsize=(12, 6))

        # Color bars based on similarity threshold
        colors = ['red' if score < 0.8 else 'blue' for score in sim_scores]

        bars = plt.bar(range(len(periods)), sim_scores, color=colors,
alpha=0.7)

        # Add value labels on bars
        for bar, score in zip(bars, sim_scores):
            plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() +
0.01,
                        f'{score:.3f}', ha='center', va='bottom',
fontweight='bold')

        plt.title("Semantic Similarity Between Consecutive Time Periods",
fontsize=16, fontweight='bold')
        plt.xlabel("Period Transitions", fontsize=12)
        plt.ylabel("Cosine Similarity", fontsize=12)
        plt.xticks(range(len(periods)), periods, rotation=45, ha='right')
        plt.grid(True, alpha=0.3, axis='y')
        plt.tight_layout()
        plt.show()
    def _plot_applicant_trends(self, top_n_applicants=5):
        """Plot applicant trends over time"""
```

```
        top_applicants =
list(self.applicant_trends.keys())[:top_n_applicants]

        fig, axes = plt.subplots(len(top_applicants), 1, figsize=(12,
4*len(top_applicants)))

        # Handle single subplot case
        if len(top_applicants) == 1:
            axes = [axes]

        for i, applicant in enumerate(top_applicants):
            applicant_data = self.applicant_trends[applicant]

            periods = list(applicant_data.keys())
            patent_counts = [applicant_data[period]['patent_count'] for
period in periods]

            axes[i].bar(periods, patent_counts, alpha=0.7, color=f'C{i}')
            axes[i].set_title(f"{applicant}", fontsize=12,
fontweight='bold')
            axes[i].set_ylabel("Patent Count")
            axes[i].grid(True, alpha=0.3, axis='y')

            # Rotate x-axis labels if needed
            if len(periods) > 5:
                axes[i].tick_params(axis='x', rotation=45)

        plt.suptitle("Patent Activity by Top Applicants Over Time",
fontsize=16, fontweight='bold')
        plt.tight_layout()
        plt.show()

    def _plot_technology_shifts(self):
        """Plot technology shift timeline"""
        if 'significant_shifts' not in self.technology_evolution:
            return

        shifts = self.technology_evolution['significant_shifts']

        if not shifts:
            print("No significant technology shifts detected")
            return

        plt.figure(figsize=(12, 6))

        # Create timeline
        for i, shift in enumerate(shifts):
            from_idx = self.time_periods.index(shift['from_period'])
            to_idx = self.time_periods.index(shift['to_period'])
```

```
            plt.plot([from_idx, to_idx],
                    [shift['similarity'], shift['similarity']],
                    'ro-', linewidth=3, markersize=8,
                    label=f"{shift['from_period']} → {shift['to_period']}")

            # Add annotation
            plt.annotate(f"{shift['similarity']:.3f}",
                        xy=((from_idx + to_idx)/2, shift['similarity']),
                        xytext=(0, 10), textcoords='offset points',
                        ha='center', fontweight='bold')

        plt.title("Significant Technology Shifts Timeline", fontsize=16,
fontweight='bold')
        plt.xlabel("Time Period", fontsize=12)
        plt.ylabel("Semantic Similarity", fontsize=12)
        plt.xticks(range(len(self.time_periods)), self.time_periods,
rotation=45)
        plt.legend()
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()

    def generate_comprehensive_report(self):
        """Generate comprehensive innovation trend analysis report"""
        print("\n" + "="*60)
        print("=== ENHANCED INNOVATION TREND ANALYSIS REPORT ===")
        print("="*60)

        # Basic statistics
        print(f"\nDATASET OVERVIEW:")
        print(f"Total patents analyzed: {len(self.data)}")
        print(f"Time period: {self.data['analysis_date'].min().strftime('%Y-
%m-%d')} to {self.data['analysis_date'].max().strftime('%Y-%m-%d')}")
        print(f"Analysis periods: {len(self.time_periods)}
({self.time_window})")
        print(f"Periods: {', '.join(self.time_periods)}")

        # Meaningful terms statistics
        all_terms = []
        for terms_list in self.data['meaningful_terms']:
            all_terms.extend(terms_list)
        unique_terms = len(set(all_terms))
        avg_terms_per_patent = len(all_terms) / len(self.data)

        print(f"\nMEANINGFUL TERMS ANALYSIS:")
        print(f"Total meaningful terms extracted: {len(all_terms)}")
        print(f"Unique meaningful terms: {unique_terms}")
        print(f"Average terms per patent: {avg_terms_per_patent:.2f}")
```

```
        # Top trending meaningful terms
        print(f"\nTOP TRENDING MEANINGFUL TERMS:")
        for i, (term, data) in
enumerate(list(self.trending_terms.items())[:15], 1):
            print(f"{i:2d}. {term:<40} (r={data['correlation']:6.3f},
p={data['p_value']:6.3f})")

        # Top declining meaningful terms
        print(f"\nTOP DECLINING MEANINGFUL TERMS:")
        for i, (term, data) in
enumerate(list(self.declining_terms.items())[:15], 1):
            print(f"{i:2d}. {term:<40} (r={data['correlation']:6.3f},
p={data['p_value']:6.3f})")

        # Most frequent meaningful terms overall
        term_counter = Counter(all_terms)
        print(f"\nMOST FREQUENT MEANINGFUL TERMS OVERALL:")
        for i, (term, count) in enumerate(term_counter.most_common(15), 1):
            frequency_per_patent = count / len(self.data)
            print(f"{i:2d}. {term:<40} ({count} occurrences,
{frequency_per_patent:.3f} per patent)")

        # Domain analysis
        self._analyze_domain_trends()

        # Semantic evolution insights
        if 'significant_shifts' in self.technology_evolution:
            shifts = self.technology_evolution['significant_shifts']
            print(f"\nSIGNIFICANT TECHNOLOGY SHIFTS:")
            if shifts:
                for shift in shifts:
                    print(f"• {shift['from_period']} → {shift['to_period']}:
Similarity = {shift['similarity']:.3f}")
            else:
                print("• No significant shifts detected (technology
evolution is gradual)")

        # Applicant insights with meaningful terms
        print(f"\nTOP APPLICANT FOCUS AREAS:")
        for applicant, trend_data in
list(self.applicant_trends.items())[:5]:
            total_patents = sum(period_data['patent_count'] for period_data
in trend_data.values())
            active_periods = len(trend_data)

            # Get most recent focus areas
            if trend_data:
                latest_period = max(trend_data.keys())
```

```
                latest_focus = trend_data[latest_period].get('focus_areas',
[])
                focus_str = ', '.join(latest_focus[:3]) if latest_focus else
'N/A'
                print(f"• {applicant}: {total_patents} patents, Recent
focus: {focus_str}")

        print("\n" + "="*60)
        print("Enhanced innovation trend analysis completed!")
        print("="*60)

    def _analyze_domain_trends(self):
        """Analyze trends by technology domain"""
        print(f"\nDOMAIN-SPECIFIC TREND ANALYSIS:")

        domain_keywords = {
            'Perception & Sensing': [
                # Core sensors
                'lidar', 'li-dar', 'light detection ranging', 'radar',
'radio detection',
                'camera system', 'vision system', 'stereo camera', 'mono
camera', 'omnidirectional camera',
                'ultrasonic sensor', 'sonar', 'time of flight', 'tof
sensor',

                # Vision techniques
                'stereo vision', 'binocular vision', 'monocular vision',
'panoramic vision',
                'depth estimation', 'depth perception', 'distance
measurement', 'range finding',
                'disparity estimation', 'triangulation',

                # Detection & Recognition
                'object detection', 'target detection', 'obstacle
detection', 'hazard detection',
                'lane detection', 'lane recognition', 'road marking
detection', 'lane boundary',
                'traffic sign recognition', 'sign detection', 'signal
recognition', 'traffic light detection',
                'pedestrian detection', 'person detection', 'human
detection', 'vulnerable road user',
                'cyclist detection', 'bicycle detection', 'bike detection',
'motorcycle detection',
                'vehicle detection', 'car detection', 'truck detection',

                # Advanced perception
                'semantic segmentation', 'pixel classification', 'scene
parsing', 'dense prediction',
```

```
                'instance segmentation', 'object segmentation', 'panoptic
segmentation',
                'point cloud', '3d point cloud', 'laser scan', 'lidar scan',
'lidar point cloud',
                'voxel grid', '3d grid', 'occupancy grid', 'occupancy map',
'grid map',

                # Multi-sensor approaches
                'sensor fusion', 'multi sensor', 'data fusion', 'information
fusion',
                'multi-modal sensing', 'multimodal perception', 'cross-
modal',
                'late fusion', 'early fusion', 'feature fusion', 'decision
fusion'
            ],

            'Localization & Mapping': [
                # SLAM variants
                'simultaneous localization mapping', 'slam', 'visual slam',
'v-slam', 'vslam',
                'lidar slam', 'l-slam', 'laser slam', 'rgb-d slam', 'stereo
slam',
                'monocular slam', 'feature based slam', 'direct slam',
'indirect slam',

                # Odometry
                'visual odometry', 'vo', 'stereo odometry', 'monocular
odometry',
                'lidar odometry', 'wheel odometry', 'inertial odometry',

                # GNSS/GPS
                'gps rtk', 'rtk gps', 'real time kinematic', 'differential
gps', 'dgps',
                'precise point positioning', 'ppp', 'carrier phase', 'code
phase',

                # Mapping
                'high definition map', 'hd map', 'high precision map',
'detailed map',
                'prior map', 'reference map', 'base map', 'semantic map',
'topological map',
                'metric map', 'feature map', 'landmark map',

                # Localization techniques
                'localization accuracy', 'pose accuracy', 'position
accuracy',
                'global localization', 'local localization',
'relocalization',
```

```
                'map matching', 'route matching', 'path matching', 'road
matching',
                'dead reckoning', 'inertial navigation', 'ins', 'integrated
navigation',

                # Filtering & estimation
                'particle filter', 'monte carlo', 'monte carlo
localization', 'mcl',
                'kalman filter', 'extended kalman', 'ekf', 'unscented
kalman', 'ukf',
                'pose estimation', 'position estimation', 'orientation
estimation',
                'state estimation', 'trajectory estimation',

                # Loop closure & optimization
                'loop closure', 'place recognition', 'loop detection',
                'pose graph optimization', 'bundle adjustment', 'graph
optimization'
            ],

            'Path Planning & Control': [
                # Planning hierarchy
                'path planning', 'route planning', 'trajectory planning',
'motion planning',
                'global planning', 'local planning', 'behavioral planning',
'strategic planning',
                'maneuver planning', 'tactical planning', 'operational
planning',

                # Optimization
                'route optimization', 'path optimization', 'trajectory
optimization',
                'multi objective optimization', 'constrained optimization',

                # Decision making
                'decision making', 'behavior planning', 'decision tree',
'finite state machine',
                'hierarchical planning', 'hybrid planning',

                # Control systems
                'steering control', 'lateral control', 'longitudinal
control', 'vehicle control',
                'throttle control', 'acceleration control', 'speed control',
'velocity control',
                'brake control', 'braking control', 'deceleration control',

                # Control algorithms
                'pid controller', 'proportional integral', 'proportional
integral derivative',
```

```
                'model predictive control', 'mpc', 'receding horizon',
                'linear quadratic', 'lqr', 'linear quadratic gaussian',
'lqg',
                'optimal control', 'robust control', 'adaptive control',

                # Safety functions
                'collision avoidance', 'obstacle avoidance', 'crash
avoidance',
                'emergency braking', 'automatic emergency', 'aeb',
'autonomous emergency braking',
                'collision mitigation', 'pre crash', 'forward collision
warning',

                # ADAS functions
                'lane keeping', 'lane centering', 'lane following', 'lane
departure warning',
                'adaptive cruise control', 'acc', 'cruise control',
'intelligent cruise control',
                'traffic jam assist', 'highway pilot', 'autopilot',

                # Trajectory execution
                'trajectory tracking', 'path following', 'reference
tracking',
                'waypoint following', 'spline following', 'curve following'
            ],

            'AI/ML for AVs': [
                # Neural network architectures
                'deep neural network', 'dnn', 'artificial neural network',
'ann',
                'convolutional neural', 'cnn', 'convnet', 'convolutional
neural network',
                'recurrent neural', 'rnn', 'recurrent neural network',
                'long short term memory', 'lstm', 'gated recurrent unit',
'gru',

                # Modern architectures
                'transformer model', 'transformer architecture', 'vision
transformer', 'vit',
                'attention mechanism', 'self attention', 'cross attention',
'multi head attention',
                'encoder decoder', 'autoencoder', 'variational autoencoder',
'vae',
                'generative adversarial', 'gan', 'generative model',
                'graph neural network', 'gnn', 'graph convolutional', 'gcn',

                # Learning paradigms
                'reinforcement learning', 'rl', 'deep reinforcement', 'drl',
'q learning',
```

```
                'policy gradient', 'actor critic', 'deep q network', 'dqn',
                'imitation learning', 'behavioral cloning', 'inverse
reinforcement',
                'end to end learning', 'e2e learning', 'end-to-end',

                # Training approaches
                'supervised learning', 'unsupervised learning', 'semi
supervised',
                'self supervised', 'contrastive learning', 'metric
learning',
                'transfer learning', 'domain adaptation', 'fine tuning',
'pre training',
                'multi task learning', 'continual learning', 'lifelong
learning',

                # Adversarial & robustness
                'adversarial training', 'adversarial example', 'robust
optimization',
                'domain randomization', 'data augmentation',
'regularization',

                # Temporal modeling
                'temporal modeling', 'sequence modeling', 'time series',
'sequential data',
                'sequence prediction', 'future prediction', 'motion
prediction',
                'trajectory prediction', 'behavior prediction', 'intent
prediction'
            ],

            'V2X Communication': [
                # V2X variants
                'vehicle to vehicle', 'v2v', 'car to car', 'c2c',
                'vehicle to infrastructure', 'v2i', 'vehicle to roadside',
'v2r',
                'vehicle to everything', 'v2x', 'vehicle to cloud', 'v2c',
                'vehicle to pedestrian', 'v2p', 'vehicle to network', 'v2n',
                'vehicle to device', 'v2d', 'vehicle to grid', 'v2g',

                # Communication technologies
                'dedicated short range', 'dsrc', 'wave', 'ieee 802.11p',
                'cellular v2x', 'c-v2x', 'lte v2x', '5g v2x', 'nr v2x',
                '5g communication', 'lte communication', 'cellular
communication',
                'wifi communication', 'bluetooth communication',

                # Cooperative behaviors
                'cooperative driving', 'coordinated driving', 'collaborative
driving',
```

```
                'cooperative perception', 'cooperative planning',
'cooperative control',
                'platooning', 'convoy', 'vehicle following', 'automated
following',
                'string stability', 'platoon control', 'cooperative adaptive
cruise',

                # Traffic coordination
                'vehicle coordination', 'traffic coordination', 'fleet
coordination',
                'intersection management', 'traffic signal', 'signal phase
timing',
                'traffic light control', 'priority request', 'preemption',
                'corridor management', 'arterial coordination',

                # Connected services
                'connected vehicle', 'connected car', 'iot vehicle',
'telematics',
                'over air update', 'ota', 'remote diagnostics', 'fleet
management',

                # Protocols & standards
                'communication protocol', 'message protocol', 'data
exchange',
                'sae j2735', 'sae j2945', 'etsi its', 'cooperative
awareness',
                'collective perception', 'maneuver coordination'
            ],

            'Safety & Validation': [
                # Functional safety standards
                'functional safety', 'safety function', 'iso 26262', 'iec
61508',
                'safety lifecycle', 'safety process', 'safety culture',

                # Safety integrity
                'safety integrity level', 'sil', 'automotive safety
integrity', 'asil',
                'asil a', 'asil b', 'asil c', 'asil d', 'safety goal',

                # Hazard analysis
                'hazard analysis', 'hazop', 'hazard operability', 'hara',
                'fault tree', 'fault tree analysis', 'fta', 'event tree
analysis',
                'failure mode', 'fmea', 'failure mode effects', 'fmeca',
                'root cause analysis', 'bow tie analysis',

                # Risk assessment
```

```
                'risk assessment', 'safety assessment', 'risk analysis',
'risk management',
                'severity', 'exposure', 'controllability', 'risk matrix',

                # Fault tolerance
                'fault tolerance', 'fault tolerant', 'error tolerance',
'failure tolerance',
                'redundancy', 'backup system', 'failover', 'graceful
degradation',
                'fail safe', 'fail operational', 'fail silent', 'safe
state',

                # Monitoring & diagnostics
                'safety monitoring', 'health monitoring', 'diagnostic
monitoring',
                'system monitoring', 'condition monitoring', 'prognostics',
                'built in test', 'self test', 'watchdog', 'plausibility
check',

                # Verification & validation
                'verification validation', 'v&v', 'testing validation',
'safety validation',
                'scenario testing', 'test scenario', 'test case', 'test
coverage',
                'corner case', 'edge case', 'boundary case', 'stress
testing',

                # Safety classification
                'safety critical', 'mission critical', 'life critical',
                'automotive safety', 'vehicle safety', 'driving safety',
'road safety'
            ],

            'Simulation & Testing': [
                # Simulation environments
                'virtual environment', 'simulation environment', 'test
environment',
                'virtual world', 'simulated world', 'digital environment',

                # Simulation platforms
                'simulation platform', 'testing platform', 'simulation
framework',
                'carla', 'airsim', 'sumo', 'prescan', 'vires vtd',
'cognata',
                'unity', 'unreal engine', 'gazebo', 'webots',

                # Digital twins
                'digital twin', 'virtual twin', 'cyber physical', 'virtual
replica',
```

```
                'digital model', 'virtual prototype', 'simulation model',

                # Data generation
                'synthetic data generation', 'artificial data', 'simulated
data',
                'procedural generation', 'automatic generation', 'data
synthesis',
                'virtual dataset', 'synthetic dataset', 'augmented data',

                # Physics simulation
                'physics simulation', 'dynamics simulation', 'vehicle
dynamics',
                'tire model', 'suspension model', 'aerodynamic model',
                'contact model', 'collision model', 'friction model',

                # Sensor simulation
                'sensor simulation', 'lidar simulation', 'camera
simulation',
                'radar simulation', 'ultrasonic simulation', 'imu
simulation',
                'sensor model', 'noise model', 'distortion model',

                # Traffic & behavior
                'traffic simulation', 'behavior simulation', 'scenario
simulation',
                'traffic flow', 'microscopic simulation', 'macroscopic
simulation',
                'agent based simulation', 'behavioral model', 'driver
model',

                # Scenario & test generation
                'scenario generation', 'test case generation', 'automatic
test generation',
                'scenario mining', 'critical scenario', 'naturalistic
scenario',
                'parametric scenario', 'logical scenario', 'concrete
scenario',

                # Testing approaches
                'test automation', 'automated testing', 'continuous
testing',
                'regression testing', 'performance testing', 'stress
testing',
                'monte carlo testing', 'statistical testing',

                # Hardware/software integration
                'hardware in loop', 'hil', 'software in loop', 'sil',
                'model in loop', 'mil', 'processor in loop', 'pil',
                'vehicle in loop', 'vil', 'driver in loop', 'dil',
```

```
                'closed loop testing', 'open loop testing', 'real time
simulation'
            ],

            'Cybersecurity for AVs': [
                # General security
                'automotive cybersecurity', 'vehicle security', 'connected
car security',
                'iot security', 'embedded security', 'system security',

                # Network security
                'can security', 'can bus security', 'automotive ethernet
security',
                'wireless security', 'cellular security', 'v2x security',
                'network security', 'communication security',

                # Threat detection
                'intrusion detection', 'anomaly detection', 'malware
detection',
                'attack detection', 'security monitoring', 'threat
monitoring',
                'behavioral analysis', 'traffic analysis', 'pattern
recognition',

                # Cryptographic security
                'encryption', 'decryption', 'cryptography', 'symmetric
encryption',
                'asymmetric encryption', 'public key infrastructure', 'pki',
                'digital signature', 'certificate', 'hash function',

                # Access control
                'authentication', 'authorization', 'access control',
'identity management',
                'multi factor authentication', 'biometric authentication',
                'role based access', 'attribute based access',

                # Secure systems
                'secure communication', 'secure boot', 'trusted execution',
                'hardware security module', 'hsm', 'trusted platform
module', 'tpm',
                'secure element', 'root of trust', 'chain of trust',

                # Security infrastructure
                'firewall', 'intrusion prevention', 'security gateway',
                'security proxy', 'vpn', 'security module',

                # Security testing
                'penetration testing', 'vulnerability assessment', 'security
audit',
```

```
                'red team', 'blue team', 'ethical hacking', 'security
testing',

                # Updates & maintenance
                'security update', 'over air', 'ota security', 'secure
update',
                'patch management', 'vulnerability management', 'security
maintenance'
            ],

            'Human-Machine Interface': [
                # Interface systems
                'human machine interface', 'hmi', 'user interface', 'driver
interface',
                'cockpit', 'dashboard', 'instrument cluster',
'infotainment',
                'head up display', 'hud', 'augmented reality', 'ar hud',

                # Monitoring systems
                'driver monitoring', 'driver attention', 'attention
monitoring',
                'drowsiness detection', 'fatigue detection', 'distraction
detection',
                'gaze tracking', 'eye tracking', 'head pose', 'facial
recognition',

                # Handover & transitions
                'takeover request', 'handover', 'mode transition',
'automation transition',
                'manual override', 'driver intervention', 'control
transition',
                'takeover time', 'reaction time', 'situation awareness',

                # User experience
                'user experience', 'ux', 'usability', 'user acceptance',
'trust',
                'human factors', 'ergonomics', 'cognitive load', 'mental
model',
                'user interaction', 'multimodal interaction', 'voice
interface',

                # ADAS levels
                'driver assistance', 'adas', 'level 0', 'level 1', 'level
2',
                'level 3', 'level 4', 'level 5', 'sae levels', 'automation
levels',
                'conditional automation', 'high automation', 'full
automation',
```

```
                # Human behavior
                'driver behavior', 'driving behavior', 'behavioral
adaptation',
                'skill degradation', 'over reliance', 'mode confusion',
                'situational awareness', 'workload', 'stress', 'comfort'
            ]
        }

        domain_trends = {}

        for domain, keywords in domain_keywords.items():
            domain_term_counts = defaultdict(int)

            # Count domain-related terms across all trending terms
            for term in self.trending_terms.keys():
                if any(keyword in term.lower() for keyword in keywords):
                    domain_term_counts[domain] += 1

            # Calculate domain trend strength
            if domain_term_counts[domain] > 0:
                # Get average correlation of domain terms
                domain_correlations = []
                for term, data in self.trending_terms.items():
                    if any(keyword in term.lower() for keyword in keywords):
                        domain_correlations.append(data['correlation'])

                if domain_correlations:
                    avg_correlation = np.mean(domain_correlations)
                    domain_trends[domain] = {
                        'trending_terms_count': domain_term_counts[domain],
                        'avg_correlation': avg_correlation
                    }

        # Sort domains by trend strength
        sorted_domains = sorted(domain_trends.items(),
                               key=lambda x: x[1]['avg_correlation'],
reverse=True)

        for domain, data in sorted_domains:
            print(f"• {domain}: {data['trending_terms_count']} trending
terms, "
                  f"avg correlation: {data['avg_correlation']:.3f}")

    def get_period_term_evolution(self, term):
        """Get detailed evolution of a specific term across periods"""
        if term not in self.trending_terms and term not in
self.declining_terms:
            return None
```

```
        # Get term data
        term_data = self.trending_terms.get(term) or
self.declining_terms.get(term)

        evolution = []
        for i, period in enumerate(self.time_periods):
            score = term_data['scores'][i]

            # Get patents containing this term in this period
            period_patents = self.period_data[period]
            containing_patents = []

            for idx, row in period_patents.iterrows():
                if term in row['meaningful_terms']:
                    containing_patents.append({
                        'title': row['invention_title_text'],
                        'applicant': row['applicant_name'],
                        'date': row['analysis_date'].strftime('%Y-%m-%d')
                    })

            evolution.append({
                'period': period,
                'frequency_score': score,
                'patent_count': len(containing_patents),
                'example_patents': containing_patents[:3]  # Show top 3
examples
            })

        return evolution

    def run_complete_analysis(self):
        """Run complete enhanced innovation trend analysis pipeline"""
        print("Starting complete enhanced innovation trend analysis...")

        # Step 1: Prepare data with meaningful term extraction
        self.prepare_temporal_corpus()

        # Step 2: Meaningful term trend analysis
        self.compute_meaningful_term_trends()

        # Step 3: Semantic analysis based on meaningful terms
        self.compute_semantic_trends()

        # Step 4: Applicant analysis
        self.analyze_applicant_trends()

        # Step 5: Generate visualizations
        self.generate_trend_visualizations()
```

```
        # Step 6: Generate comprehensive report
        self.generate_comprehensive_report()

        print("Complete enhanced innovation trend analysis finished!")

        return {
            'trending_meaningful_terms': self.trending_terms,
            'declining_meaningful_terms': self.declining_terms,
            'applicant_trends': self.applicant_trends,
            'technology_evolution': self.technology_evolution,
            'time_periods': self.time_periods,
            'meaningful_terms_stats': {
                'total_unique_terms': len(set([term for terms_list in
self.data['meaningful_terms']
                                              for term in terms_list])),
                'avg_terms_per_patent': np.mean([len(terms_list) for
terms_list in self.data['meaningful_terms']])
            }
        }

# Example usage:
if __name__ == "__main__":
    # Initialize analyzer
    analyzer = EnhancedInnovationTrendAnalyzer(
        file_path='/content/av_patentdata.jsonl',
        time_window='yearly',
        min_patents_per_period=10
    )

    # Run complete analysis
    results = analyzer.run_complete_analysis()

    # Example: Get detailed evolution of a specific term
    # evolution = analyzer.get_period_term_evolution('machine learning')
    # if evolution:
    #     print(f"\nEvolution of 'machine learning':")
    #     for period_data in evolution:
    #         print(f"{period_data['period']}: {period_data['patent_count']}
patents")
```

Technology Emergence Analysis:

```
import json
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
from typing import Dict, List, Tuple, Optional, Any
import warnings
warnings.filterwarnings('ignore')
```

```
# Core ML and time series libraries
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
from sklearn.model_selection import TimeSeriesSplit, cross_val_score
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Time series specific
try:
    from prophet import Prophet
    PROPHET_AVAILABLE = True
except ImportError:
    PROPHET_AVAILABLE = False
    print("Prophet not available. Install with: pip install prophet")

# Deep learning for LSTM (optional)
try:
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Dropout
    from tensorflow.keras.optimizers import Adam
    KERAS_AVAILABLE = True
except ImportError:
    KERAS_AVAILABLE = False
    # Remove the print statement to avoid showing this error

# Additional libraries
from collections import Counter, defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from itertools import combinations

class TechnologyEmergencePredictor:
    """
    A comprehensive system for predicting technology emergence in autonomous
vehicle patents.

    Features:
    - Multiple forecasting models (Prophet, LSTM, ensemble methods)
    - Rich feature engineering from patent data
    - Technology growth analysis and ranking
    - Emergence event detection
    - Competitive landscape insights
    """

    def __init__(self, data_path: str = None, patents_data: List[Dict] =
None):
        """
```

```
        Initialize the predictor with patent data.

        Args:
            data_path: Path to JSON file containing patent data
            patents_data: Direct list of patent dictionaries
        """
        self.patents_data = None
        self.df = None
        self.exploded_df = None  # Add this
        self.time_series_data = None
        self.models = {}
        self.predictions = {}
        self.feature_importance = {}
        self.emergence_scores = None

        if data_path:
            self.load_data(data_path)
        elif patents_data:
            self.patents_data = patents_data
            self.preprocess_data()

    def load_data(self, data_path: str):
        """Load patent data from JSON file."""
        with open(data_path, 'r', encoding='utf-8') as f:
            self.patents_data = json.load(f)
        print(f"Loaded {len(self.patents_data)} patents from {data_path}")
        self.preprocess_data()

    def preprocess_data(self):
        """Convert patent data to structured DataFrame with enhanced
features."""
        print("Preprocessing patent data...")
        records = []

        for patent in self.patents_data:
            # Extract basic information
            record = {
                'lens_id': patent.get('lens_id'),
                'applicant_name': patent.get('applicant_name', ''),
                'date_published':
pd.to_datetime(patent.get('date_published'), errors='coerce'),
                'earliest_claim_date':
pd.to_datetime(patent.get('earliest_claim_date'), errors='coerce'),
                'primary_category': patent.get('primary_category', ''),
                'categorization_confidence':
patent.get('categorization_confidence', 'Low'),
                'estimated_tokens': patent.get('estimated_tokens', 0)
            }
```

```
            # Process AV technology areas
            av_areas = patent.get('av_technology_areas', [])
            if isinstance(av_areas, list):
                record['av_tech_areas'] = av_areas
            else:
                record['av_tech_areas'] = [av_areas] if av_areas else []
            record['num_av_areas'] = len(record['av_tech_areas'])

            # Process secondary categories
            secondary_cats = patent.get('secondary_categories', [])
            if isinstance(secondary_cats, list):
                record['secondary_categories'] = secondary_cats
            else:
                record['secondary_categories'] = [secondary_cats] if
secondary_cats else []
            record['num_secondary_categories'] =
len(record['secondary_categories'])

            # Process CPC symbols
            cpc_symbols = patent.get('cpc_symbols', [])
            if isinstance(cpc_symbols, list):
                record['cpc_symbols'] = cpc_symbols
            else:
                record['cpc_symbols'] = [cpc_symbols] if cpc_symbols else []
            record['num_cpc_symbols'] = len(record['cpc_symbols'])

            # Process claims
            claims = patent.get('claims', [])
            if isinstance(claims, list):
                record['num_claims'] = len(claims)
                record['avg_claim_length'] =
np.mean([len(str(claim).split()) for claim in claims]) if claims else 0
            else:
                record['num_claims'] = 1 if claims else 0
                record['avg_claim_length'] = len(str(claims).split()) if
claims else 0

            # Text complexity features
            record['title_length'] =
len(str(patent.get('invention_title_text', '')).split())
            record['abstract_length'] = len(str(patent.get('abstract_text',
'')).split())
            record['description_length'] = len(str(patent.get('description',
'')).split())

            # Time features
            if record['date_published'] and record['earliest_claim_date']:
                record['filing_to_pub_days'] = (record['date_published'] -
record['earliest_claim_date']).days
```

```
            else:
                record['filing_to_pub_days'] = 0

            records.append(record)

        self.df = pd.DataFrame(records)
        print(f"Created DataFrame with {len(self.df)} records")

        # Debug: Check for empty av_tech_areas
        empty_av_areas = self.df[self.df['num_av_areas'] == 0]
        if len(empty_av_areas) > 0:
            print(f"Warning: {len(empty_av_areas)} patents have no AV
technology areas")

        self._create_time_series_features()

    def _create_time_series_features(self):
        """Create time series data for each technology area."""
        print("Creating time series features...")

        # Use earliest_claim_date as the primary date for innovation timing
        self.df['date'] =
self.df['earliest_claim_date'].fillna(self.df['date_published'])

        # Remove rows with no valid dates
        initial_count = len(self.df)
        self.df = self.df.dropna(subset=['date'])
        print(f"Removed {initial_count - len(self.df)} patents with no valid
dates")

        if len(self.df) == 0:
            print("ERROR: No patents with valid dates found!")
            return

        # Create monthly time series
        self.df['year_month'] = self.df['date'].dt.to_period('M')

        # Explode AV technology areas to create one row per patent-
technology combination
        exploded_data = []
        for _, row in self.df.iterrows():
            av_areas = row['av_tech_areas']
            if not av_areas:  # If no AV areas, use primary category
                av_areas = [row['primary_category']] if
row['primary_category'] else ['Unknown']

            for tech_area in av_areas:
                if tech_area and tech_area.strip():  # Only add non-empty
tech areas
```

```
                    new_row = row.copy()
                    new_row['technology'] = tech_area.strip()
                    exploded_data.append(new_row)

        if not exploded_data:
            print("ERROR: No valid technology areas found!")
            return

        self.exploded_df = pd.DataFrame(exploded_data)
        print(f"Created exploded DataFrame with {len(self.exploded_df)}
technology-patent pairs")

        # Print technology distribution
        tech_counts = self.exploded_df['technology'].value_counts()
        print(f"Found {len(tech_counts)} unique technologies")
        print("Top 10 technologies by patent count:")
        print(tech_counts.head(10))

        # Create time series aggregations
        self.time_series_data = self._create_aggregated_time_series()
        print(f"Created time series for {len(self.time_series_data)}
technologies")

    def _create_aggregated_time_series(self) -> Dict[str, pd.DataFrame]:
        """Fixed version of time series aggregation."""
        tech_time_series = {}

        if self.exploded_df is None or len(self.exploded_df) == 0:
            print("ERROR: No exploded data available!")
            return {}

        # CRITICAL FIX: Use 'technology' field from exploded data, not
'primary_category'
        all_technologies = self.exploded_df['technology'].unique()
        print(f"Processing {len(all_technologies)} unique technologies from
exploded data")

        # Use the 'date' field that was set in _create_time_series_features
        if 'date' not in self.exploded_df.columns:
            print("ERROR: 'date' column missing from exploded_df")
            return {}

        # Ensure date is datetime
        self.exploded_df['date'] = pd.to_datetime(self.exploded_df['date'])

        min_date = self.exploded_df['date'].min()
        max_date = self.exploded_df['date'].max()

        print(f"Date range: {min_date} to {max_date}")
```

```
        # Create monthly date range
        date_range = pd.date_range(start=min_date, end=max_date,
freq='MS')  # Month start
        print(f"Created date range with {len(date_range)} months")

        for tech in all_technologies:
            # CRITICAL FIX: Filter by 'technology' field, not
'primary_category'
            tech_data = self.exploded_df[self.exploded_df['technology'] ==
tech].copy()

            if len(tech_data) == 0:
                print(f"WARNING: No data found for technology '{tech}'")
                continue

            print(f"Processing {tech}: {len(tech_data)} patents")

            # Create 'year_month' from 'date' for proper grouping
            tech_data['year_month'] =
tech_data['date'].dt.to_period('M').dt.to_timestamp()

            # Group by year_month and aggregate
            monthly_stats = tech_data.groupby('year_month').agg({
                'lens_id': 'count',                # patent count
                'applicant_name': 'nunique',      # unique applicants
                'num_claims': ['mean', 'sum'],
                'estimated_tokens': ['mean', 'sum'],
                'num_cpc_symbols': 'mean',
                'categorization_confidence': lambda x: (x == 'High').sum() /
len(x) if len(x) > 0 else 0
            }).reset_index()

            # Flatten columns properly
            monthly_stats.columns = [
                'date', 'patent_count', 'unique_applicants',
                'avg_claims', 'total_claims', 'avg_tokens', 'total_tokens',
                'avg_cpc_symbols', 'high_confidence_ratio'
            ]

            print(f"  Aggregated to {len(monthly_stats)} months with data")
            print(f"  Total patents in aggregation:
{monthly_stats['patent_count'].sum()}")

            # Complete date range with zero filling
            complete_series = pd.DataFrame({'date': date_range})
            monthly_stats = complete_series.merge(monthly_stats, on='date',
how='left').fillna(0)
```

```
            # Add derived features
            monthly_stats['technology'] = tech
            monthly_stats['cumulative_patents'] =
monthly_stats['patent_count'].cumsum()
            monthly_stats['rolling_3m_avg'] =
monthly_stats['patent_count'].rolling(3, min_periods=1).mean()
            monthly_stats['rolling_6m_avg'] =
monthly_stats['patent_count'].rolling(6, min_periods=1).mean()
            monthly_stats['patent_velocity'] =
monthly_stats['patent_count'].diff().fillna(0)
            monthly_stats['applicant_diversity'] =
monthly_stats['unique_applicants'] / (monthly_stats['patent_count'] + 1)

            # Calculate market concentration
            monthly_stats['market_concentration'] =
self._calculate_market_concentration(tech_data, monthly_stats)

            final_patent_count = monthly_stats['patent_count'].sum()
            print(f"  {tech}: {final_patent_count} patents across
{len(monthly_stats)} months")

            if final_patent_count > 0:  # Only add if we have actual patents
                tech_time_series[tech] = monthly_stats
            else:
                print(f"  WARNING: Skipping {tech} - no patents in final
aggregation")

        print(f"Successfully created time series for {len(tech_time_series)}
technologies")
        return tech_time_series

    def _calculate_market_concentration(self, tech_data: pd.DataFrame,
monthly_stats: pd.DataFrame) -> pd.Series:
        """Calculate market concentration using Herfindahl index
approximation."""
        concentration_scores = []

        for _, month_row in monthly_stats.iterrows():
            month_start = month_row['date']
            month_end = month_start + pd.DateOffset(months=1)

            # Use the correct date column
            month_patents = tech_data[
                (tech_data['date'] >= month_start) &
                (tech_data['date'] < month_end)
            ]

            if len(month_patents) == 0:
                concentration_scores.append(0)
```

```
                continue

            # Calculate market shares by applicant
            applicant_counts =
month_patents['applicant_name'].value_counts()
            if len(applicant_counts) == 0:
                concentration_scores.append(0)
                continue

            market_shares = applicant_counts / applicant_counts.sum()
            herfindahl_index = (market_shares ** 2).sum()
            concentration_scores.append(herfindahl_index)

        return pd.Series(concentration_scores)

    def debug_data_flow(self):
        """Debug method to trace data through the pipeline."""
        print("=== DATA FLOW DEBUG ===")

        if self.df is not None:
            print(f"Main DataFrame: {len(self.df)} rows")
            print(f"Date range: {self.df['date'].min()} to
{self.df['date'].max()}")
            print(f"Primary categories:
{self.df['primary_category'].nunique()} unique")
            print("Sample primary categories:",
self.df['primary_category'].value_counts().head(3).to_dict())

        if self.exploded_df is not None:
            print(f"Exploded DataFrame: {len(self.exploded_df)} rows")
            print(f"Technologies: {self.exploded_df['technology'].nunique()}
unique")
            print("Sample technologies:",
self.exploded_df['technology'].value_counts().head(3).to_dict())

        if self.time_series_data:
            print(f"Time series data: {len(self.time_series_data)}
technologies")
            for tech_name, tech_df in
list(self.time_series_data.items())[:3]:
                total_patents = tech_df['patent_count'].sum()
                print(f"  {tech_name}: {total_patents} total patents,
{len(tech_df)} time points")

        print("=== END DEBUG ===")

    def engineer_features(self, tech_df: pd.DataFrame) -> pd.DataFrame:
        """Engineer additional predictive features for a technology time
series."""
```

```
        features_df = tech_df.copy()

        # Lag features
        for lag in [1, 2, 3, 6, 12]:
            if len(features_df) > lag:
                features_df[f'patent_count_lag_{lag}'] =
features_df['patent_count'].shift(lag)
                features_df[f'unique_applicants_lag_{lag}'] =
features_df['unique_applicants'].shift(lag)

        # Rolling statistics
        for window in [3, 6, 12]:
            if len(features_df) >= window:
                features_df[f'patent_count_rolling_std_{window}'] =
features_df['patent_count'].rolling(window).std()
                features_df[f'patent_count_rolling_max_{window}'] =
features_df['patent_count'].rolling(window).max()
                features_df[f'unique_applicants_rolling_mean_{window}'] =
features_df['unique_applicants'].rolling(window).mean()

        # Trend features
        features_df['patent_count_trend'] =
features_df['patent_count'].rolling(6, min_periods=3).apply(
            lambda x: np.polyfit(range(len(x)), x, 1)[0] if len(x) >= 2 else
0
        )

        # Seasonal features
        features_df['month'] = features_df['date'].dt.month
        features_df['quarter'] = features_df['date'].dt.quarter
        features_df['year'] = features_df['date'].dt.year

        # Innovation intensity features
        features_df['innovation_intensity'] = (
            features_df['patent_count'] * features_df['avg_tokens'] *
features_df['high_confidence_ratio']
        )

        # Competition features
        features_df['competitive_pressure'] =
features_df['unique_applicants'] / (features_df['patent_count'] + 1)

        return features_df.fillna(0)

    def train_prophet_model(self, tech_name: str, forecast_periods: int =
12) -> Dict[str, Any]:
        """Train Prophet model for a specific technology."""
        if not PROPHET_AVAILABLE:
```

```
            raise ImportError("Prophet not available. Install with: pip
install prophet")

        tech_df = self.time_series_data[tech_name].copy()

        # Prepare data for Prophet
        prophet_df = tech_df[['date',
'patent_count']].rename(columns={'date': 'ds', 'patent_count': 'y'})

        # Add additional regressors
        prophet_df['unique_applicants'] = tech_df['unique_applicants']
        prophet_df['avg_tokens'] = tech_df['avg_tokens']
        prophet_df['market_concentration'] = tech_df['market_concentration']

        # Initialize and fit Prophet model
        model = Prophet(
            yearly_seasonality=True,
            monthly_seasonality=True,
            changepoint_prior_scale=0.05,
            seasonality_prior_scale=10.0
        )

        # Add regressors
        model.add_regressor('unique_applicants')
        model.add_regressor('avg_tokens')
        model.add_regressor('market_concentration')

        model.fit(prophet_df)

        # Create future dataframe
        future = model.make_future_dataframe(periods=forecast_periods,
freq='M')

        # Add regressor values for future periods (using last known values)
        last_values = prophet_df.iloc[-1]
        future['unique_applicants'] =
future['unique_applicants'].fillna(last_values['unique_applicants'])
        future['avg_tokens'] =
future['avg_tokens'].fillna(last_values['avg_tokens'])
        future['market_concentration'] =
future['market_concentration'].fillna(last_values['market_concentration'])

        # Make predictions
        forecast = model.predict(future)

        return {
            'model': model,
            'forecast': forecast,
            'train_data': prophet_df,
```

```
            'future_periods': forecast_periods
        }

    def train_ml_ensemble(self, tech_name: str, forecast_periods: int = 12)
-> Dict[str, Any]:
        """Train ensemble ML models for technology prediction."""
        tech_df = self.time_series_data[tech_name].copy()
        features_df = self.engineer_features(tech_df)

        # Prepare features and target
        feature_cols = [col for col in features_df.columns if col not in [
            'date', 'year_month', 'technology', 'patent_count'
        ]]

        X = features_df[feature_cols].fillna(0)
        y = features_df['patent_count']

        # Time series split for validation
        tscv = TimeSeriesSplit(n_splits=3)

        # Models
        models = {
            'random_forest': RandomForestRegressor(n_estimators=100,
random_state=42),
            'gradient_boosting': GradientBoostingRegressor(n_estimators=100,
random_state=42)
        }

        trained_models = {}
        feature_importance = {}

        for name, model in models.items():
            # Cross-validation
            scores = cross_val_score(model, X, y, cv=tscv,
scoring='neg_mean_absolute_error')

            # Fit on full data
            model.fit(X, y)
            trained_models[name] = model

            # Feature importance
            if hasattr(model, 'feature_importances_'):
                importance_df = pd.DataFrame({
                    'feature': feature_cols,
                    'importance': model.feature_importances_
                }).sort_values('importance', ascending=False)
                feature_importance[name] = importance_df

        # Create predictions
```

```
        last_features = X.iloc[-1:].copy()
        predictions = {}

        for name, model in trained_models.items():
            model_predictions = []
            current_features = last_features.copy()

            for _ in range(forecast_periods):
                pred = model.predict(current_features)[0]
                model_predictions.append(max(0, pred))  # Ensure non-
negative

                # Update features for next prediction (simple approach)
                current_features = current_features.copy()
                # This is a simplified feature update - in practice, you'd
want more sophisticated logic

            predictions[name] = model_predictions

        # Ensemble prediction (average)
        ensemble_prediction = np.mean([predictions[name] for name in
predictions], axis=0)

        return {
            'models': trained_models,
            'predictions': predictions,
            'ensemble_prediction': ensemble_prediction,
            'feature_importance': feature_importance,
            'feature_columns': feature_cols
        }
    def calculate_emergence_scores(self) -> pd.DataFrame:
        """Calculate comprehensive emergence scores for all technologies."""
        print("Calculating emergence scores...")

        if not self.time_series_data:
            print("ERROR: No time series data available!")
            return pd.DataFrame()

        emergence_data = []

        for tech_name, tech_df in self.time_series_data.items():
            print(f"Processing {tech_name}: {len(tech_df)} data points")

            # Relaxed minimum data requirements
            if len(tech_df) < 2:
                print(f"  Skipping {tech_name}: insufficient data points
({len(tech_df)})")
                continue
```

```
            # Filter out rows with zero patent counts for better analysis
            active_data = tech_df[tech_df['patent_count'] > 0]
            if len(active_data) < 1:
                print(f"  Skipping {tech_name}: no active periods")
                continue

            print(f"  {tech_name}: {len(active_data)} active periods")

            # Split data into periods for growth calculation
            total_months = len(tech_df)
            split_point = max(1, total_months // 2)

            earlier_data = tech_df.iloc[:split_point]
            recent_data = tech_df.iloc[split_point:]

            # Growth metrics - use sum instead of mean for more meaningful
comparison
            recent_sum = recent_data['patent_count'].sum()
            earlier_sum = earlier_data['patent_count'].sum()

            # Normalize by time periods to get rate per month
            recent_avg = recent_sum / len(recent_data) if len(recent_data) >
0 else 0
            earlier_avg = earlier_sum / len(earlier_data) if
len(earlier_data) > 0 else 0

            # Calculate growth rate with better handling
            if earlier_avg > 0:
                growth_rate = (recent_avg - earlier_avg) / earlier_avg
            elif recent_avg > 0:
                growth_rate = 1.0  # 100% growth from zero baseline
            else:
                growth_rate = 0

            # Acceleration (second derivative)
            patent_counts = active_data['patent_count'].values
            if len(patent_counts) >= 3:
                # Calculate acceleration more robustly
                velocity = np.diff(patent_counts)
                acceleration = np.mean(np.diff(velocity)) if len(velocity) >
1 else 0
            else:
                acceleration = 0

            # Diversity metrics - use recent data
            recent_applicant_diversity =
recent_data['applicant_diversity'].mean()
            recent_market_concentration =
recent_data['market_concentration'].mean()
```

```
            # Innovation quality - fix the token calculation
            recent_avg_tokens = recent_data['avg_tokens'].mean()
            if recent_avg_tokens == 0 or np.isnan(recent_avg_tokens):
                # Fallback to total tokens divided by patents
                recent_avg_tokens = (recent_data['total_tokens'].sum() /
                                   max(recent_data['patent_count'].sum(),
1))

            recent_confidence = recent_data['high_confidence_ratio'].mean()

            # Volatility (coefficient of variation) - use active data only
            if len(patent_counts) > 1 and np.mean(patent_counts) > 0:
                volatility = np.std(patent_counts) / np.mean(patent_counts)
            else:
                volatility = 0

            # Trend strength - use active data
            x = np.arange(len(patent_counts))
            if len(patent_counts) >= 2:
                slope, intercept, r_value, p_value, std_err =
stats.linregress(x, patent_counts)
                trend_strength = abs(r_value)
                trend_significance = 1 - p_value if p_value < 0.05 else 0
            else:
                trend_strength = 0
                trend_significance = 0

            # Composite emergence score with better normalization
            # Normalize growth rate to [0, 1] range
            normalized_growth = min(max(growth_rate, -1), 2) / 3 + 1/3  #
Maps [-1, 2] to [0, 1]

            # Normalize acceleration
            normalized_acceleration = min(max(acceleration, -1), 1) / 2 +
0.5  # Maps [-1, 1] to [0, 1]

            # Normalize token count (assume reasonable range 0-2000)
            normalized_tokens = min(recent_avg_tokens / 2000, 1)

            emergence_score = (
                0.30 * normalized_growth +
                0.20 * normalized_acceleration +
                0.15 * recent_applicant_diversity +
                0.15 * (1 - recent_market_concentration) +
                0.10 * normalized_tokens +
                0.10 * recent_confidence
            )
```

```
            emergence_data.append({
                'technology': tech_name,
                'emergence_score': emergence_score,
                'growth_rate': growth_rate,
                'acceleration': acceleration,
                'recent_avg_patents': recent_avg,
                'applicant_diversity': recent_applicant_diversity,
                'market_concentration': recent_market_concentration,
                'avg_innovation_quality': recent_avg_tokens,
                'confidence_ratio': recent_confidence,
                'trend_strength': trend_strength,
                'trend_significance': trend_significance,
                'volatility': volatility,
                'total_patents': tech_df['patent_count'].sum(),
                'data_points': len(tech_df)
            })

        if not emergence_data:
            print("WARNING: No emergence data calculated!")
            return pd.DataFrame()

        emergence_df = pd.DataFrame(emergence_data)
        emergence_df = emergence_df.sort_values('emergence_score',
ascending=False)

        print(f"Calculated emergence scores for {len(emergence_df)}
technologies")
        self.emergence_scores = emergence_df
        return emergence_df

    def predict_all_technologies(self, forecast_periods: int = 12, methods:
List[str] = None) -> Dict[str, Dict]:
        """Run predictions for all technologies using specified methods."""
        if methods is None:
            methods = ['ensemble']
            if PROPHET_AVAILABLE:
                methods.append('prophet')

        all_predictions = {}

        for tech_name in self.time_series_data.keys():
            tech_predictions = {}

            if 'prophet' in methods and PROPHET_AVAILABLE:
                try:
                    prophet_result = self.train_prophet_model(tech_name,
forecast_periods)
                    tech_predictions['prophet'] = prophet_result
                except Exception as e:
```

```
                    print(f"Prophet failed for {tech_name}: {e}")

            if 'ensemble' in methods:
                try:
                    ensemble_result = self.train_ml_ensemble(tech_name,
forecast_periods)
                    tech_predictions['ensemble'] = ensemble_result
                except Exception as e:
                    print(f"Ensemble failed for {tech_name}: {e}")

            if tech_predictions:
                all_predictions[tech_name] = tech_predictions

        self.predictions = all_predictions
        return all_predictions

    def get_top_emerging_technologies(self, top_n: int = 10) ->
pd.DataFrame:
        """Get top N emerging technologies based on comprehensive
scoring."""
        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            print("Calculating emergence scores...")
            self.calculate_emergence_scores()

        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            print("WARNING: No emergence scores available!")
            return pd.DataFrame()

        return self.emergence_scores.head(top_n)

    def generate_insights_report(self) -> Dict[str, Any]:
        """Generate comprehensive insights report."""
        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            self.calculate_emergence_scores()

        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            return {
                'error': 'No emergence scores available',
                'summary': {
                    'total_technologies': len(self.time_series_data) if
self.time_series_data else 0,
                    'total_patents': len(self.df) if self.df is not None
else 0,
                }
            }

        report = {
            'summary': {
                'total_technologies': len(self.time_series_data),
```

```
                'total_patents': self.df.shape[0],
                'date_range': {
                    'start': str(self.df['date'].min().date()),
                    'end': str(self.df['date'].max().date())
                },
                'top_applicants':
self.df['applicant_name'].value_counts().head(10).to_dict()
            },
            'emergence_analysis': {
                'highest_growth_technologies':
self.emergence_scores.nlargest(5, 'growth_rate')[
                    ['technology', 'growth_rate', 'emergence_score']
                ].to_dict('records'),
                'most_diverse_technologies':
self.emergence_scores.nlargest(5, 'applicant_diversity')[
                    ['technology', 'applicant_diversity', 'emergence_score']
                ].to_dict('records'),
                'highest_quality_innovations':
self.emergence_scores.nlargest(5, 'avg_innovation_quality')[
                    ['technology', 'avg_innovation_quality',
'emergence_score']
                ].to_dict('records')
            },
            'market_dynamics': {
                'most_concentrated_markets':
self.emergence_scores.nlargest(5, 'market_concentration')[
                    ['technology', 'market_concentration']
                ].to_dict('records'),
                'most_competitive_markets':
self.emergence_scores.nsmallest(5, 'market_concentration')[
                    ['technology', 'market_concentration',
'applicant_diversity']
                ].to_dict('records')
            }
        }

        return report

    def visualize_emergence_landscape(self, save_path: str = None):
        """Create comprehensive visualization of the technology emergence
landscape."""
        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            print("No emergence scores available, calculating...")
            self.calculate_emergence_scores()

        if self.emergence_scores is None or len(self.emergence_scores) == 0:
            print("ERROR: Cannot create visualization - no emergence scores
available!")
            return
```

```
        fig, axes = plt.subplots(2, 2, figsize=(16, 12))
        fig.suptitle('Technology Emergence Landscape', fontsize=16,
fontweight='bold')

        # 1. Emergence Score vs Growth Rate
        ax1 = axes[0, 0]
        plot_data = self.emergence_scores[
            (self.emergence_scores['growth_rate'] >= -2) &
            (self.emergence_scores['growth_rate'] <= 5) &
            (self.emergence_scores['avg_innovation_quality'] > 0)
        ]

        if len(plot_data) > 0:
            scatter = ax1.scatter(
                plot_data['growth_rate'],
                plot_data['emergence_score'],
                s=plot_data['total_patents'] * 3 + 20,
                alpha=0.7,
                c=plot_data['applicant_diversity'],
                cmap='viridis'
            )
            ax1.set_xlabel('Growth Rate')
            ax1.set_ylabel('Emergence Score')
            ax1.set_title('Emergence Score vs Growth Rate\n(Size = Total
Patents, Color = Applicant Diversity)')
            plt.colorbar(scatter, ax=ax1)
        else:
            ax1.text(0.5, 0.5, 'No data available for plotting',
                     ha='center', va='center', transform=ax1.transAxes)

        # 2. Top 10 Emerging Technologies
        ax2 = axes[0, 1]
        top_10 = self.emergence_scores.head(10)
        bars = ax2.barh(range(len(top_10)), top_10['emergence_score'],
color='skyblue')
        ax2.set_yticks(range(len(top_10)))
        ax2.set_yticklabels([tech[:20] + '...' if len(tech) > 20 else tech
for tech in top_10['technology']])
        ax2.set_xlabel('Emergence Score')
        ax2.set_title('Top 10 Emerging Technologies')
        ax2.invert_yaxis()

        # 3. Market Concentration vs Innovation Quality
        ax3 = axes[1, 0]
        ax3.scatter(
            self.emergence_scores['market_concentration'],
            self.emergence_scores['avg_innovation_quality'],
            s=self.emergence_scores['emergence_score'] * 100,
```

```python
            alpha=0.6,
            color='purple'
        )
        ax3.set_xlabel('Market Concentration')
        ax3.set_ylabel('Average Innovation Quality (Tokens)')
        ax3.set_title('Market Concentration vs Innovation Quality\n(Size =
Emergence Score)')

        # 4. Technology Portfolio Quadrant Analysis
        ax4 = axes[1, 1]
        median_growth = self.emergence_scores['growth_rate'].median()
        median_diversity =
self.emergence_scores['applicant_diversity'].median()

        colors = []
        for _, row in self.emergence_scores.iterrows():
            if row['growth_rate'] >= median_growth and
row['applicant_diversity'] >= median_diversity:
                colors.append('green')
            elif row['growth_rate'] >= median_growth:
                colors.append('orange')
            elif row['applicant_diversity'] >= median_diversity:
                colors.append('blue')
            else:
                colors.append('red')

        ax4.scatter(
            self.emergence_scores['growth_rate'],
            self.emergence_scores['applicant_diversity'],
            c=colors,
            alpha=0.6
        )
        ax4.axvline(median_growth, color='black', linestyle='--', alpha=0.5)
        ax4.axhline(median_diversity, color='black', linestyle='--',
alpha=0.5)
        ax4.set_xlabel('Growth Rate')
        ax4.set_ylabel('Applicant Diversity')
        ax4.set_title('Technology Portfolio Quadrant Analysis')

        plt.tight_layout(rect=[0, 0.03, 1, 0.95])

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')
        plt.show()

# Example usage and testing functions
def example_usage():
    """Example of how to use the TechnologyEmergencePredictor class."""
```

```
    # Initialize predictor (assuming you have the JSON file)
    predictor = TechnologyEmergencePredictor('av_patent_data.json')

    # Calculate emergence scores
    emergence_scores = predictor.calculate_emergence_scores()
    if not emergence_scores.empty:
        print(emergence_scores[['technology', 'emergence_score',
'growth_rate']].head(10))
    else:
        print("No emergence scores available.")
    print("Top 10 Emerging Technologies:")
    print(emergence_scores[['technology', 'emergence_score',
'growth_rate']].head(10))

    # Run predictions for all technologies
    predictions = predictor.predict_all_technologies(forecast_periods=12,
methods=['ensemble'])

    # Generate insights report
    report = predictor.generate_insights_report()
    print("\nInsights Report Summary:")
    print(f"Total Technologies Analyzed:
{report['summary']['total_technologies']}")
    print(f"Total Patents: {report['summary']['total_patents']}")
    print(f"Date Range: {report['summary']['date_range']['start']} to
{report['summary']['date_range']['end']}")

    # Visualize results
    predictor.visualize_emergence_landscape("emergence.png")

    return predictor, emergence_scores, predictions, report

if __name__ == "__main__":
    # Run example if this file is executed directly
    example_usage()
```

Innovation Outlier Detection:

```
import json
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.decomposition import PCA
from sklearn.cluster import DBSCAN
from sklearn.metrics.pairwise import cosine_similarity
from scipy import stats
```

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')

class InnovationOutlierDetector:
    """
    Advanced system for detecting innovation outliers in patent data.
    Identifies radical innovations, strategic pivots, and hidden gems
through
    multi-dimensional anomaly detection combining textual, temporal, and
metadata features.
    """

    def __init__(self, data_path=None, data=None):
        """
        Initialize the outlier detector.

        Args:
            data_path (str): Path to JSON file containing patent data
            data (list): Direct patent data as list of dictionaries
        """
        self.data_path = data_path
        self.raw_data = data
        self.df = None
        self.feature_vectors = None
        self.outlier_scores = {}
        self.models = {}
        self.scalers = {}
        self.analysis_results = {}

    def load_and_preprocess_data(self):
        """Load and preprocess patent data for outlier detection."""
        if self.raw_data is None:
            with open(self.data_path, 'r', encoding='utf-8') as f:
                self.raw_data = json.load(f)

        # Convert to DataFrame
        self.df = pd.DataFrame(self.raw_data)

        # Clean and preprocess dates
        self.df['date_published'] =
pd.to_datetime(self.df['date_published'], errors='coerce')
```

```
        self.df['earliest_claim_date'] = 
pd.to_datetime(self.df['earliest_claim_date'], errors='coerce')

        # Calculate patent age and time gaps
        current_date = datetime.now()
        self.df['patent_age_days'] = (current_date - 
self.df['date_published']).dt.days
        self.df['claim_to_pub_gap'] = (self.df['date_published'] - 
self.df['earliest_claim_date']).dt.days

        # Handle missing values
        self.df['abstract_text'] = self.df['abstract_text'].fillna('')
        self.df['claims'] = self.df['claims'].fillna('')
        self.df['description'] = self.df['description'].fillna('')

        print(f"Loaded {len(self.df)} patents for outlier analysis")

    def extract_textual_features(self):
        """Extract advanced textual features for novelty detection."""
        features = {}

        # 1. Textual Uniqueness Features
        print("Extracting textual uniqueness features...")

        # TF-IDF for different text fields
        tfidf_abstract = TfidfVectorizer(max_features=500, 
stop_words='english', ngram_range=(1,2))
        tfidf_claims = TfidfVectorizer(max_features=500, 
stop_words='english', ngram_range=(1,2))

        abstract_vectors = 
tfidf_abstract.fit_transform(self.df['abstract_text'])

        # Fix: Convert claims list to string properly
        claims_text = self.df['claims'].apply(
            lambda x: ' '.join(x) if isinstance(x, list) else str(x) if x is 
not None else ''
        )
        claims_vectors = tfidf_claims.fit_transform(claims_text)

        # Calculate uniqueness scores based on cosine similarity
        features['abstract_uniqueness'] = 
self._calculate_uniqueness_scores(abstract_vectors)
        features['claims_uniqueness'] = 
self._calculate_uniqueness_scores(claims_vectors)

        # 2. Language Complexity Features
        features['abstract_length'] = self.df['abstract_text'].str.len()
```

```
        # Fix: Handle claims list properly for length calculation
        features['claims_length'] = claims_text.str.len()
        features['description_length'] = self.df['description'].str.len()

        # Technical term density
        technical_terms = ['algorithm', 'neural', 'machine learning',
'artificial intelligence',
                          'deep learning', 'computer vision', 'sensor
fusion', 'lidar', 'radar',
                          'autonomous', 'automated', 'self-driving',
'perception', 'localization']

        features['technical_density'] = self.df['abstract_text'].apply(
            lambda x: sum(term in str(x).lower() for term in
technical_terms) / max(len(str(x).split()), 1)
        )

        # 3. Novelty Language Indicators
        novelty_indicators = ['novel', 'innovative', 'breakthrough',
'revolutionary', 'unprecedented',
                            'first', 'unique', 'advanced', 'improved',
'enhanced', 'optimized']

        features['novelty_language_score'] = self.df['abstract_text'].apply(
            lambda x: sum(term in str(x).lower() for term in
novelty_indicators) / max(len(str(x).split()), 1)
        )

        return features

    def extract_metadata_features(self):
        """Extract metadata-based outlier features."""
        features = {}

        print("Extracting metadata features...")

        # 1. Temporal Outliers
        features['patent_age_days'] = self.df['patent_age_days']
        features['claim_to_pub_gap'] = self.df['claim_to_pub_gap'].fillna(0)

        # Publication timing anomalies (patents published at unusual times)
        self.df['pub_year'] = self.df['date_published'].dt.year
        self.df['pub_month'] = self.df['date_published'].dt.month

        # Calculate z-scores for publication timing
        features['pub_timing_anomaly'] =
np.abs(stats.zscore(self.df['pub_month'].fillna(6)))

        # 2. CPC Classification Rarity
```

```
        # Flatten CPC symbols and calculate rarity scores
        all_cpc = []
        for cpc_list in self.df['cpc_symbols']:
            if isinstance(cpc_list, list):
                all_cpc.extend(cpc_list)

        cpc_counts = pd.Series(all_cpc).value_counts()

        # Calculate rarity score for each patent
        features['cpc_rarity_score'] = self.df['cpc_symbols'].apply(
            lambda x: np.mean([1/cpc_counts.get(cpc, 1) for cpc in x if
isinstance(x, list) and cpc in cpc_counts])
            if isinstance(x, list) else 0
        )

        # 3. Applicant Strategic Shift Detection
        # Group patents by applicant and analyze technology diversity
        applicant_tech_diversity =
self.df.groupby('applicant_name')['primary_category'].nunique()
        features['applicant_diversity'] =
self.df['applicant_name'].map(applicant_tech_diversity)

        # 4. Technology Area Concentration
        features['tech_area_count'] = self.df['av_technology_areas'].apply(
            lambda x: len(x) if isinstance(x, list) else 0
        )

        # Note: Removed confidence_score and confidence_anomaly as they're
not useful for outlier detection
        # (LLM categorization confidence is uniform across patents)

        return features

    def extract_innovation_patterns(self):
        """Extract advanced innovation pattern features."""
        features = {}

        print("Extracting innovation pattern features...")

        # 1. Cross-Domain Innovation Detection
        # Patents that span multiple traditionally separate domains
        traditional_domains = {
            'hardware': ['sensor', 'camera', 'lidar', 'radar', 'hardware'],
            'software': ['algorithm', 'software', 'code', 'program'],
            'ai_ml': ['artificial intelligence', 'machine learning',
'neural', 'deep learning'],
            'communication': ['communication', 'v2x', 'network',
'connectivity'],
            'safety': ['safety', 'security', 'cybersecurity', 'protection']
```

```
        }

        domain_scores = {}
        for domain, keywords in traditional_domains.items():
            domain_scores[domain] = self.df['abstract_text'].apply(
                lambda x: sum(keyword in str(x).lower() for keyword in
keywords)
            )

        # Calculate cross-domain innovation score
        domain_matrix = pd.DataFrame(domain_scores)
        features['cross_domain_score'] = (domain_matrix > 0).sum(axis=1)
        features['domain_diversity'] = domain_matrix.std(axis=1)

        # 2. Invention Intensity - Fix: Ensure proper numeric types
        description_length = self.df['description'].str.len().fillna(0)
        estimated_tokens = pd.to_numeric(self.df['estimated_tokens'],
errors='coerce').fillna(1)

        features['invention_intensity'] = (
            description_length * features['cross_domain_score'] /
            (estimated_tokens + 1)
        )

        # 3. Problem-Solution Novelty Gap - Fix: Handle string length
properly
        problem_complexity =
self.df['problem_addressed'].str.len().fillna(0)
        solution_novelty = self.df['novelty_aspect'].str.len().fillna(0)
        features['novelty_gap'] = solution_novelty / (problem_complexity +
1)

        return features

    def _calculate_uniqueness_scores(self, vectors):
        """Calculate uniqueness scores based on cosine similarity."""
        # Calculate pairwise similarities
        similarities = cosine_similarity(vectors)

        # For each patent, calculate average similarity to all others
        # Lower similarity = higher uniqueness
        avg_similarities = similarities.mean(axis=1)
        uniqueness_scores = 1 - avg_similarities

        return uniqueness_scores

    def build_autoencoder_detector(self, feature_matrix, encoding_dim=32):
        """Build autoencoder for anomaly detection."""
        input_dim = feature_matrix.shape[1]
```

```
        # Encoder
        input_layer = Input(shape=(input_dim,))
        encoded = Dense(128, activation='relu')(input_layer)
        encoded = Dropout(0.2)(encoded)
        encoded = Dense(64, activation='relu')(encoded)
        encoded = Dropout(0.2)(encoded)
        encoded = Dense(encoding_dim, activation='relu')(encoded)

        # Decoder
        decoded = Dense(64, activation='relu')(encoded)
        decoded = Dropout(0.2)(decoded)
        decoded = Dense(128, activation='relu')(decoded)
        decoded = Dropout(0.2)(decoded)
        decoded = Dense(input_dim, activation='linear')(decoded)

        # Autoencoder model
        autoencoder = Model(input_layer, decoded)
        autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

        return autoencoder

    def detect_outliers(self):
        """Main method to detect innovation outliers using multiple
techniques."""
        print("Starting comprehensive outlier detection...")

        # Extract all feature types
        textual_features = self.extract_textual_features()
        metadata_features = self.extract_metadata_features()
        pattern_features = self.extract_innovation_patterns()

        # Combine all features
        all_features = {**textual_features, **metadata_features,
**pattern_features}
        feature_df = pd.DataFrame(all_features)

        # Handle missing values
        feature_df = feature_df.fillna(feature_df.median())

        # Standardize features
        scaler = StandardScaler()
        feature_matrix = scaler.fit_transform(feature_df)
        self.scalers['main'] = scaler

        # 1. Isolation Forest Detection
        print("Running Isolation Forest detection...")
        iso_forest = IsolationForest(contamination=0.1, random_state=42,
n_estimators=200)
```

```
        iso_outliers = iso_forest.fit_predict(feature_matrix)
        iso_scores = iso_forest.decision_function(feature_matrix)

        self.outlier_scores['isolation_forest'] = {
            'predictions': iso_outliers,
            'scores': iso_scores
        }

        # 2. Autoencoder Detection
        print("Running Autoencoder detection...")
        autoencoder = self.build_autoencoder_detector(feature_matrix)
        autoencoder.fit(feature_matrix, feature_matrix,
                       epochs=50, batch_size=32, verbose=0,
validation_split=0.2)

        # Calculate reconstruction errors
        reconstructed = autoencoder.predict(feature_matrix, verbose=0)
        reconstruction_errors = np.mean(np.square(feature_matrix -
reconstructed), axis=1)

        # Determine autoencoder outliers (top 10% reconstruction errors)
        error_threshold = np.percentile(reconstruction_errors, 90)
        autoencoder_outliers = (reconstruction_errors >
error_threshold).astype(int)
        autoencoder_outliers[autoencoder_outliers == 0] = -1  # Convert to -
1/1 format

        self.outlier_scores['autoencoder'] = {
            'predictions': autoencoder_outliers,
            'scores': reconstruction_errors
        }

        # 3. DBSCAN Clustering for Outlier Detection
        print("Running DBSCAN clustering...")
        dbscan = DBSCAN(eps=0.5, min_samples=5)
        cluster_labels = dbscan.fit_predict(feature_matrix)

        # Points labeled as -1 are outliers in DBSCAN
        dbscan_outliers = (cluster_labels == -1).astype(int)
        dbscan_outliers[dbscan_outliers == 0] = -1

        self.outlier_scores['dbscan'] = {
            'predictions': dbscan_outliers,
            'scores': cluster_labels
        }

        # 4. Statistical Outlier Detection (Multivariate)
        print("Running statistical outlier detection...")
        # Mahalanobis distance
```

```
        try:
            cov_matrix = np.cov(feature_matrix.T)
            inv_cov_matrix = np.linalg.inv(cov_matrix)
            mean_vec = np.mean(feature_matrix, axis=0)

            mahal_distances = []
            for row in feature_matrix:
                diff = row - mean_vec
                mahal_dist = np.sqrt(np.dot(np.dot(diff, inv_cov_matrix),
diff))
                mahal_distances.append(mahal_dist)

            mahal_distances = np.array(mahal_distances)
            mahal_threshold = np.percentile(mahal_distances, 90)
            statistical_outliers = (mahal_distances >
mahal_threshold).astype(int)
            statistical_outliers[statistical_outliers == 0] = -1

            self.outlier_scores['statistical'] = {
                'predictions': statistical_outliers,
                'scores': mahal_distances
            }
        except:
            print("Mahalanobis distance calculation failed, using z-score
method")
            z_scores = np.abs(stats.zscore(feature_matrix, axis=0))
            max_z_scores = np.max(z_scores, axis=1)
            z_threshold = 2.5
            statistical_outliers = (max_z_scores > z_threshold).astype(int)
            statistical_outliers[statistical_outliers == 0] = -1

            self.outlier_scores['statistical'] = {
                'predictions': statistical_outliers,
                'scores': max_z_scores
            }

        # Store feature information
        self.feature_vectors = feature_matrix
        self.feature_names = list(feature_df.columns)

        print("Outlier detection completed!")

    def ensemble_outlier_scoring(self):
        """Combine multiple outlier detection methods for robust scoring."""
        print("Creating ensemble outlier scores...")

        # Normalize all scores to [0, 1] range
        normalized_scores = {}
```

```
        for method, results in self.outlier_scores.items():
            scores = results['scores']
            if method == 'dbscan':
                # For DBSCAN, convert cluster labels to outlier scores
                scores = (scores == -1).astype(float)
            else:
                # Normalize scores to [0, 1]
                scores = (scores - scores.min()) / (scores.max() -
scores.min())

            normalized_scores[method] = scores

        # Create ensemble score (weighted average)
        weights = {
            'isolation_forest': 0.3,
            'autoencoder': 0.3,
            'statistical': 0.25,
            'dbscan': 0.15
        }

        ensemble_scores = np.zeros(len(self.df))
        for method, weight in weights.items():
            if method in normalized_scores:
                ensemble_scores += weight * normalized_scores[method]

        # Create final outlier predictions
        ensemble_threshold = np.percentile(ensemble_scores, 85)  # Top 15%
as outliers
        ensemble_predictions = (ensemble_scores >
ensemble_threshold).astype(int)

        self.outlier_scores['ensemble'] = {
            'predictions': ensemble_predictions,
            'scores': ensemble_scores
        }

        return ensemble_scores, ensemble_predictions

    def analyze_outlier_characteristics(self):
        """Analyze characteristics of detected outliers."""
        print("Analyzing outlier characteristics...")

        ensemble_scores, ensemble_predictions =
self.ensemble_outlier_scoring()

        # Add outlier information to dataframe
        self.df['outlier_score'] = ensemble_scores
        self.df['is_outlier'] = ensemble_predictions
```

```
        # Separate outliers and normal patents
        outliers = self.df[self.df['is_outlier'] == 1].copy()
        normal = self.df[self.df['is_outlier'] == 0].copy()

        analysis = {}

        # 1. Outlier Statistics
        analysis['outlier_count'] = len(outliers)
        analysis['outlier_percentage'] = (len(outliers) / len(self.df)) *
100

        # 2. Applicant Analysis
        outlier_applicants =
outliers['applicant_name'].value_counts().head(10)
        normal_applicants = normal['applicant_name'].value_counts().head(10)

        analysis['top_outlier_applicants'] = outlier_applicants.to_dict()
        analysis['applicant_outlier_rates'] = {}

        for applicant in outlier_applicants.index:
            total_patents = len(self.df[self.df['applicant_name'] ==
applicant])
            outlier_patents = len(outliers[outliers['applicant_name'] ==
applicant])
            analysis['applicant_outlier_rates'][applicant] =
(outlier_patents / total_patents) * 100

        # 3. Technology Area Analysis
        outlier_tech_areas = []
        for areas in outliers['av_technology_areas']:
            if isinstance(areas, list):
                outlier_tech_areas.extend(areas)

        analysis['outlier_tech_distribution'] =
pd.Series(outlier_tech_areas).value_counts().to_dict()

        # 4. Temporal Analysis
        outliers['pub_year'] = outliers['date_published'].dt.year
        analysis['outlier_temporal_distribution'] =
outliers['pub_year'].value_counts().sort_index().to_dict()

        # 5. Novel Language Analysis - Fix: Handle string length properly
        analysis['avg_outlier_novelty_score'] =
outliers['novelty_aspect'].str.len().mean()
        analysis['avg_normal_novelty_score'] =
normal['novelty_aspect'].str.len().mean()

        self.analysis_results = analysis
```

```
        return analysis

    def identify_strategic_pivots(self):
        """Identify companies making strategic pivots based on outlier
patterns."""
        print("Identifying strategic pivots...")

        pivots = {}

        # Group by applicant and analyze outlier patterns over time
        for applicant in self.df['applicant_name'].unique():
            applicant_data = self.df[self.df['applicant_name'] ==
applicant].copy()

            if len(applicant_data) < 5:  # Skip applicants with too few
patents
                continue

            # Sort by date
            applicant_data = applicant_data.sort_values('date_published')

            # Calculate rolling outlier rate
            applicant_data['year'] =
applicant_data['date_published'].dt.year
            yearly_outliers =
applicant_data.groupby('year')['is_outlier'].agg(['count', 'sum'])
            yearly_outliers['outlier_rate'] = yearly_outliers['sum'] /
yearly_outliers['count']

            # Detect significant increases in outlier rate
            if len(yearly_outliers) >= 3:
                outlier_rates = yearly_outliers['outlier_rate'].values
                rate_changes = np.diff(outlier_rates)

                # Look for sustained increases
                if np.any(rate_changes > 0.3):  # 30% increase in outlier
rate
                    pivot_year =
yearly_outliers.index[np.argmax(rate_changes) + 1]

                    pivots[applicant] = {
                        'pivot_year': pivot_year,
                        'outlier_rate_increase': np.max(rate_changes),
                        'total_patents': len(applicant_data),
                        'outlier_patents':
applicant_data['is_outlier'].sum(),
                        'recent_tech_areas':
applicant_data[applicant_data['year'] >=
pivot_year]['primary_category'].value_counts().to_dict()
```

```
                        }

        return pivots

    def generate_insights_report(self):
        """Generate comprehensive insights report."""
        print("Generating insights report...")

        # Ensure analysis is complete
        if not self.outlier_scores:
            self.detect_outliers()

        analysis = self.analyze_outlier_characteristics()
        pivots = self.identify_strategic_pivots()

        # Get top outliers for detailed analysis
        top_outliers = self.df.nlargest(20, 'outlier_score')[
            ['lens_id', 'invention_title_text', 'applicant_name',
'date_published',
             'primary_category', 'outlier_score', 'novelty_aspect']
        ]

        report = {
            'summary': {
                'total_patents_analyzed': len(self.df),
                'outliers_detected': analysis['outlier_count'],
                'outlier_percentage': analysis['outlier_percentage'],
                'detection_methods_used': list(self.outlier_scores.keys())
            },
            'top_outlier_patents': top_outliers.to_dict('records'),
            'strategic_pivots': pivots,
            'outlier_characteristics': analysis,
            'key_insights': self._generate_key_insights(analysis, pivots,
top_outliers)
        }

        return report

    def _generate_key_insights(self, analysis, pivots, top_outliers):
        """Generate key insights from the analysis."""
        insights = []

        # Innovation leaders
        if analysis['top_outlier_applicants']:
            top_innovator =
list(analysis['top_outlier_applicants'].keys())[0]
            insights.append(f"Most innovative outlier producer:
{top_innovator} with {analysis['top_outlier_applicants'][top_innovator]}
radical innovations")
```

```
        # Emerging technologies
        if analysis['outlier_tech_distribution']:
            top_tech = max(analysis['outlier_tech_distribution'],
key=analysis['outlier_tech_distribution'].get)
            insights.append(f"Technology area with most radical innovations:
{top_tech}")

        # Strategic pivots
        if pivots:
            pivot_companies = list(pivots.keys())[:3]
            insights.append(f"Companies showing strategic pivots: {',
'.join(pivot_companies)}")

        # Temporal patterns
        if analysis['outlier_temporal_distribution']:
            peak_year = max(analysis['outlier_temporal_distribution'],
key=analysis['outlier_temporal_distribution'].get)
            insights.append(f"Peak year for radical innovations:
{peak_year}")

        # Novelty patterns
        if 'avg_outlier_novelty_score' in analysis:
            novelty_ratio = analysis['avg_outlier_novelty_score'] /
analysis['avg_normal_novelty_score']
            insights.append(f"Outlier patents have {novelty_ratio:.2f}x more
detailed novelty descriptions")

        return insights

    def visualize_outliers(self, save_plots=True):
        """Create visualizations for outlier analysis."""
        plt.style.use('default')
        fig, axes = plt.subplots(2, 3, figsize=(18, 12))
        fig.suptitle('Innovation Outlier Detection Analysis', fontsize=16,
fontweight='bold')

        # 1. Outlier Score Distribution
        axes[0, 0].hist(self.df['outlier_score'], bins=50, alpha=0.7,
color='skyblue', edgecolor='black')
        axes[0, 0].axvline(self.df['outlier_score'].quantile(0.85),
color='red', linestyle='--', label='Outlier Threshold')
        axes[0, 0].set_xlabel('Outlier Score')
        axes[0, 0].set_ylabel('Frequency')
        axes[0, 0].set_title('Distribution of Outlier Scores')
        axes[0, 0].legend()

        # 2. Top Outlier Applicants
        if self.analysis_results['top_outlier_applicants']:
```

```
                top_applicants =
list(self.analysis_results['top_outlier_applicants'].items())[:8]
                companies, counts = zip(*top_applicants)
                axes[0, 1].barh(range(len(companies)), counts,
color='lightcoral')
                axes[0, 1].set_yticks(range(len(companies)))
                axes[0, 1].set_yticklabels([c[:20] + '...' if len(c) > 20 else c
for c in companies])
                axes[0, 1].set_xlabel('Number of Outlier Patents')
                axes[0, 1].set_title('Top Companies by Outlier Patents')

            # 3. Temporal Distribution
            if self.analysis_results['outlier_temporal_distribution']:
                years =
list(self.analysis_results['outlier_temporal_distribution'].keys())
                counts =
list(self.analysis_results['outlier_temporal_distribution'].values())
                axes[0, 2].plot(years, counts, marker='o', linewidth=2,
markersize=6, color='green')
                axes[0, 2].set_xlabel('Year')
                axes[0, 2].set_ylabel('Number of Outliers')
                axes[0, 2].set_title('Outlier Patents Over Time')
                axes[0, 2].grid(True, alpha=0.3)

            # 4. Technology Area Distribution
            if self.analysis_results['outlier_tech_distribution']:
                tech_areas =
list(self.analysis_results['outlier_tech_distribution'].items())[:8]
                areas, counts = zip(*tech_areas)
                axes[1, 0].pie(counts, labels=[a[:15] + '...' if len(a) > 15
else a for a in areas],
                              autopct='%1.1f%%', startangle=90)
                axes[1, 0].set_title('Outlier Distribution by Technology Area')

            # 5. Outlier Score vs Patent Age
            axes[1, 1].scatter(self.df['patent_age_days'],
self.df['outlier_score'],
                              alpha=0.6, c=self.df['is_outlier'], cmap='RdYlBu')
            axes[1, 1].set_xlabel('Patent Age (Days)')
            axes[1, 1].set_ylabel('Outlier Score')
            axes[1, 1].set_title('Outlier Score vs Patent Age')

            # 6. Method Comparison
            methods = ['isolation_forest', 'autoencoder', 'statistical',
'dbscan']
            method_counts = []
            for method in methods:
                if method in self.outlier_scores:
```

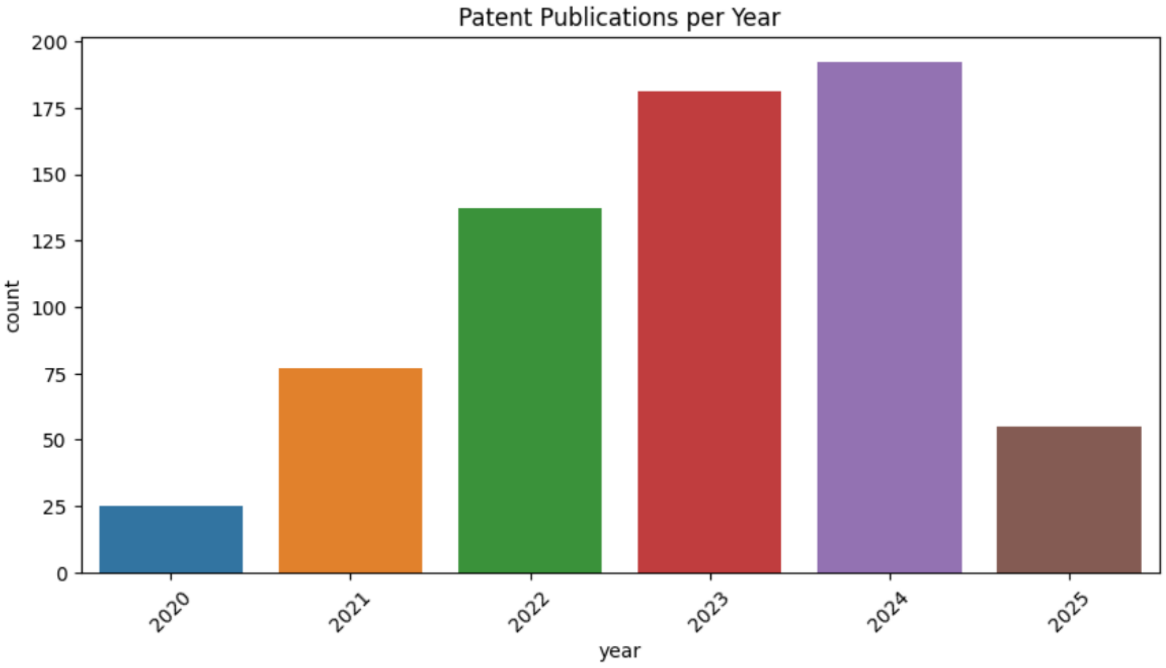
```
                outlier_count = 
np.sum(self.outlier_scores[method]['predictions'] == 1)
                method_counts.append(outlier_count)
            else:
                method_counts.append(0)

        axes[1, 2].bar(methods, method_counts, color=['blue', 'orange', 
'green', 'red'], alpha=0.7)
        axes[1, 2].set_xlabel('Detection Method')
        axes[1, 2].set_ylabel('Outliers Detected')
        axes[1, 2].set_title('Outliers by Detection Method')
        axes[1, 2].tick_params(axis='x', rotation=45)

        plt.tight_layout()

        if save_plots:
            plt.savefig('innovation_outlier_analysis.png', dpi=300, 
bbox_inches='tight')
            print("Plots saved as 'innovation_outlier_analysis.png'")

        plt.show()

        return fig

    def export_results(self, filename='innovation_outliers_results.json'):
        """Export analysis results to JSON file."""
        results = self.generate_insights_report()

        # Convert numpy types to Python types for JSON serialization
        def convert_numpy(obj):
            if isinstance(obj, np.integer):
                return int(obj)
            elif isinstance(obj, np.floating):
                return float(obj)
            elif isinstance(obj, np.ndarray):
                return obj.tolist()
            return obj

        # Deep convert all numpy types
        def deep_convert(obj):
            if isinstance(obj, dict):
                return {k: deep_convert(v) for k, v in obj.items()}
            elif isinstance(obj, list):
                return [deep_convert(v) for v in obj]
            else:
                return convert_numpy(obj)

        results = deep_convert(results)
```

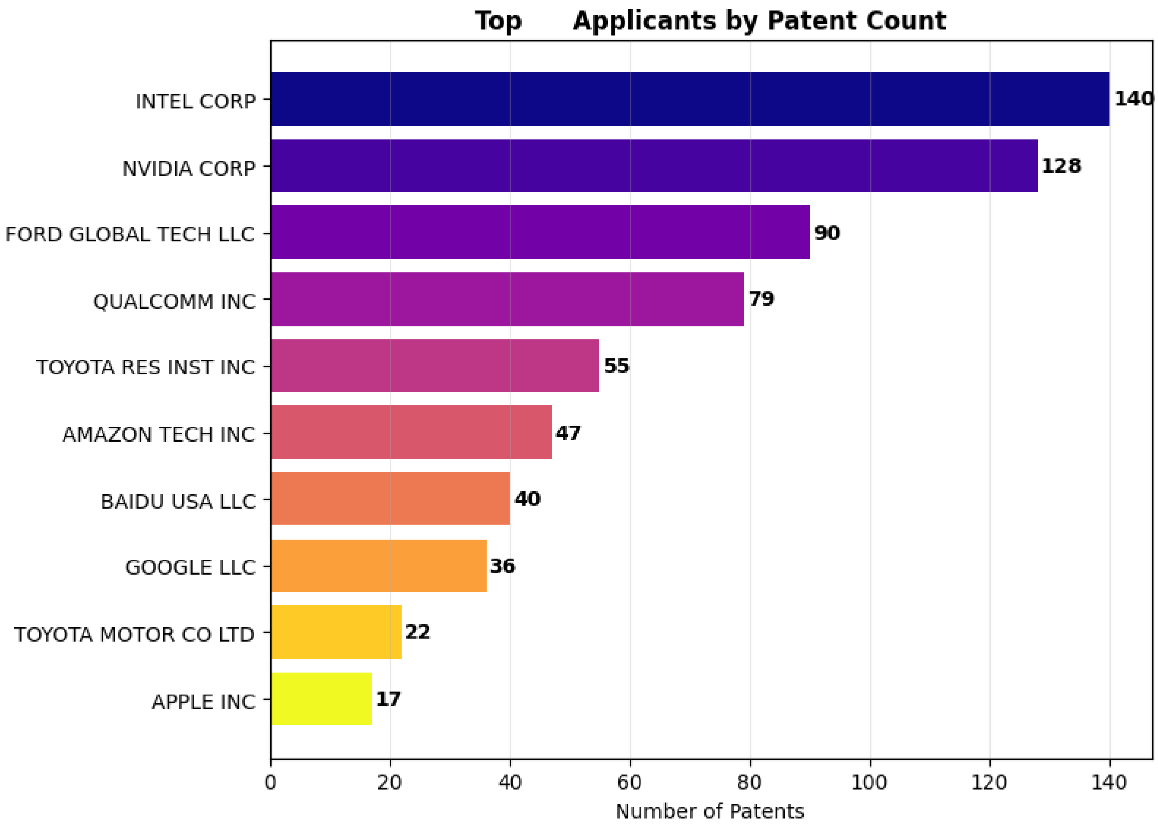
```
        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(results, f, indent=2, ensure_ascii=False, default=str)

        print(f"Results exported to {filename}")

        # Also export detailed outlier data
        outlier_data = self.df[self.df['is_outlier'] == 1][
            ['lens_id', 'invention_title_text', 'applicant_name',
'date_published',
             'primary_category', 'outlier_score', 'abstract_text',
'novelty_aspect']
        ].copy()

        outlier_filename = 'detailed_outlier_patents.csv'
        outlier_data.to_csv(outlier_filename, index=False)
        print(f"Detailed outlier patents exported to {outlier_filename}")

    def run_complete_analysis(self, visualize=True, export=True):
        """Run the complete innovation outlier detection pipeline."""
        print("="*60)
        print("INNOVATION OUTLIER DETECTION PIPELINE")
        print("="*60)

        # Step 1: Load and preprocess data
        self.load_and_preprocess_data()

        # Step 2: Detect outliers using multiple methods
        self.detect_outliers()

        # Step 3: Generate comprehensive analysis
        report = self.generate_insights_report()

        # Step 4: Print summary
        self._print_summary_report(report)

        # Step 5: Visualize results
        if visualize:
            self.visualize_outliers()

        # Step 6: Export results
        if export:
            self.export_results()

        print("="*60)
        print("ANALYSIS COMPLETE!")
        print("="*60)

        return report
```

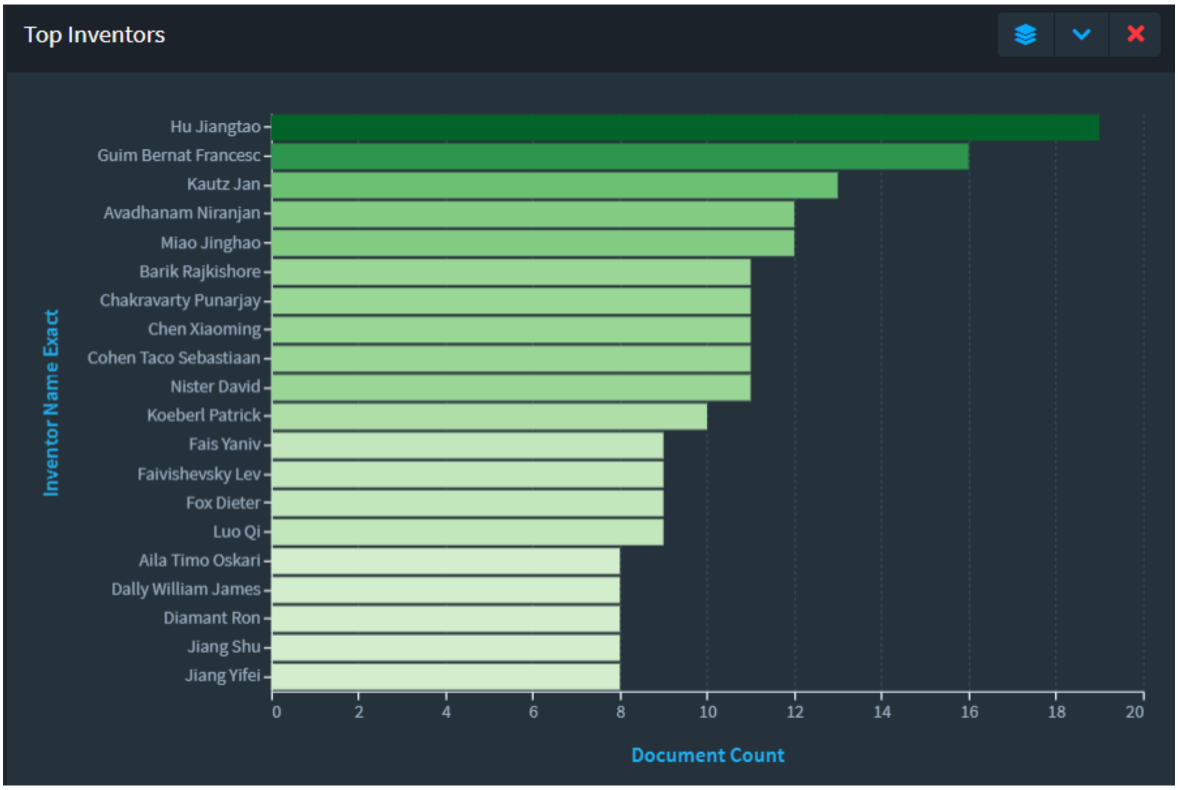
```
    def _print_summary_report(self, report):
        """Print a formatted summary report."""
        print("\n" + "="*50)
        print("INNOVATION OUTLIER DETECTION SUMMARY")
        print("="*50)

        # Summary statistics
        summary = report['summary']
        print(f"\n ANALYSIS OVERVIEW:")
        print(f"   Total Patents Analyzed:
{summary['total_patents_analyzed']:,}")
        print(f"   Outliers Detected: {summary['outliers_detected']:,}")
        print(f"   Outlier Percentage:
{summary['outlier_percentage']:.2f}%")
        print(f"   Detection Methods: {',
'.join(summary['detection_methods_used'])}")

        # Top outlier patents
        print(f"\n TOP 5 RADICAL INNOVATION PATENTS:")
        for i, patent in enumerate(report['top_outlier_patents'][:5], 1):
            print(f"   {i}. {patent['invention_title_text'][:60]}...")
            print(f"      Company: {patent['applicant_name']}")
            print(f"      Outlier Score: {patent['outlier_score']:.4f}")
            print(f"      Technology: {patent['primary_category']}")
            print()

        # Strategic pivots
        if report['strategic_pivots']:
            print(f" STRATEGIC PIVOTS DETECTED
({len(report['strategic_pivots'])} companies):")
            for company, pivot_info in
list(report['strategic_pivots'].items())[:3]:
                print(f"   • {company}")
                print(f"     Pivot Year: {pivot_info['pivot_year']}")
                print(f"     Outlier Rate Increase:
{pivot_info['outlier_rate_increase']:.1%}")
                print(f"     Recent Focus:
{list(pivot_info['recent_tech_areas'].keys())[:2]}")
                print()

        # Key insights
        print(f" KEY INSIGHTS:")
        for insight in report['key_insights']:
            print(f"   • {insight}")

        # Innovation leaders
        outlier_chars = report['outlier_characteristics']
        if outlier_chars['top_outlier_applicants']:
            print(f"\n TOP INNOVATION LEADERS:")
```

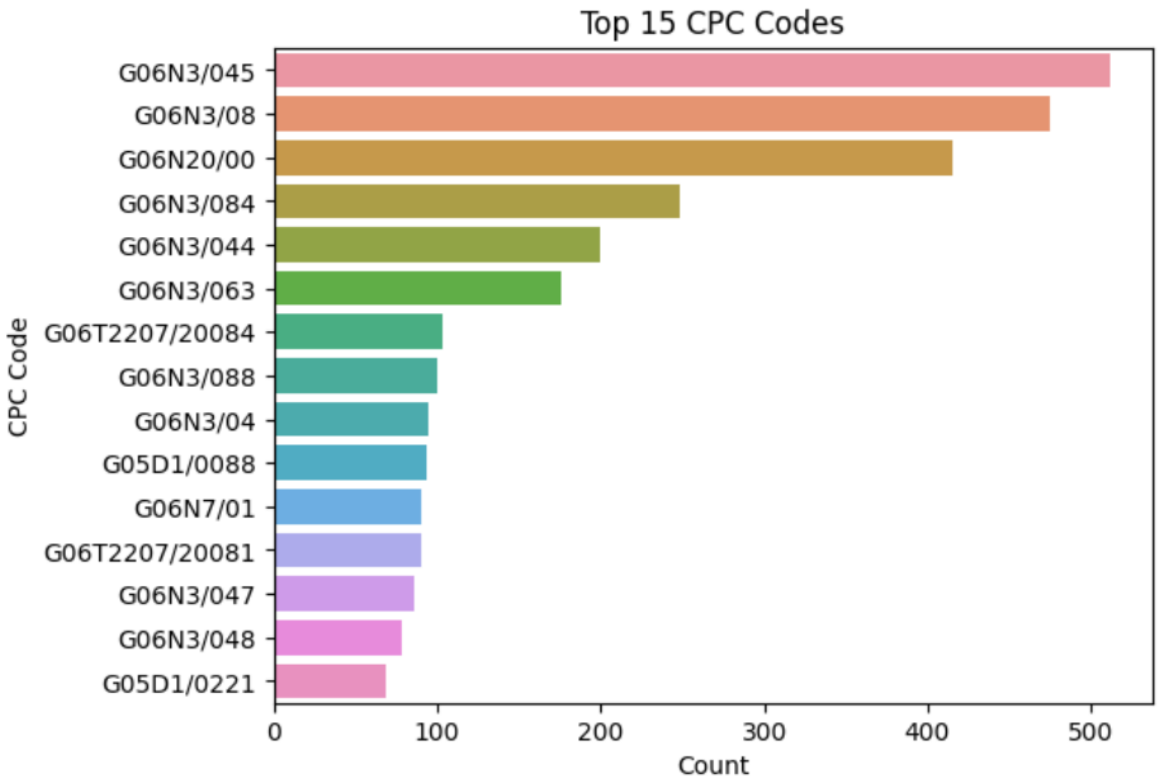
```
            for company, count in
list(outlier_chars['top_outlier_applicants'].items())[:5]:
                outlier_rate =
outlier_chars['applicant_outlier_rates'].get(company, 0)
                print(f"    • {company}: {count} outlier patents
({outlier_rate:.1f}% rate)")

        print("\n" + "="*50)

# Example usage and demonstration
if __name__ == "__main__":
    # Uncomment below to run with your data:
    detector = InnovationOutlierDetector('/content/av_patent_data.json')
    results = detector.run_complete_analysis()
```

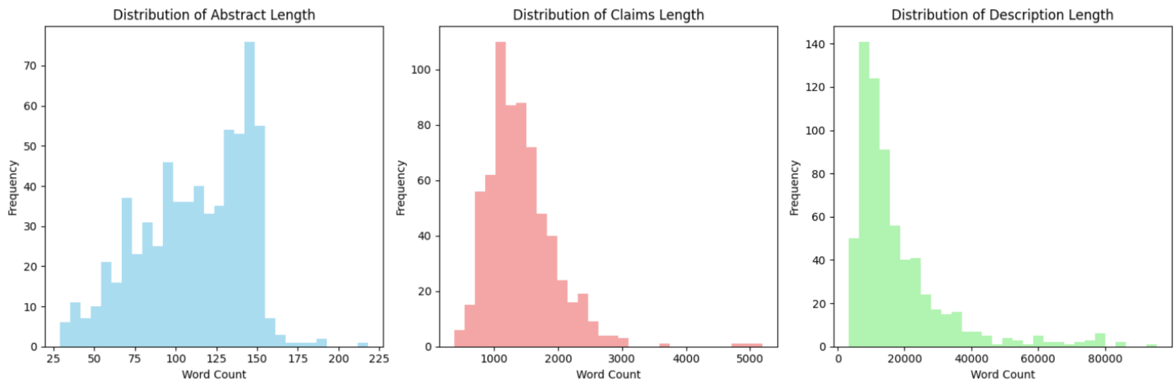
SCREENSHOTS:

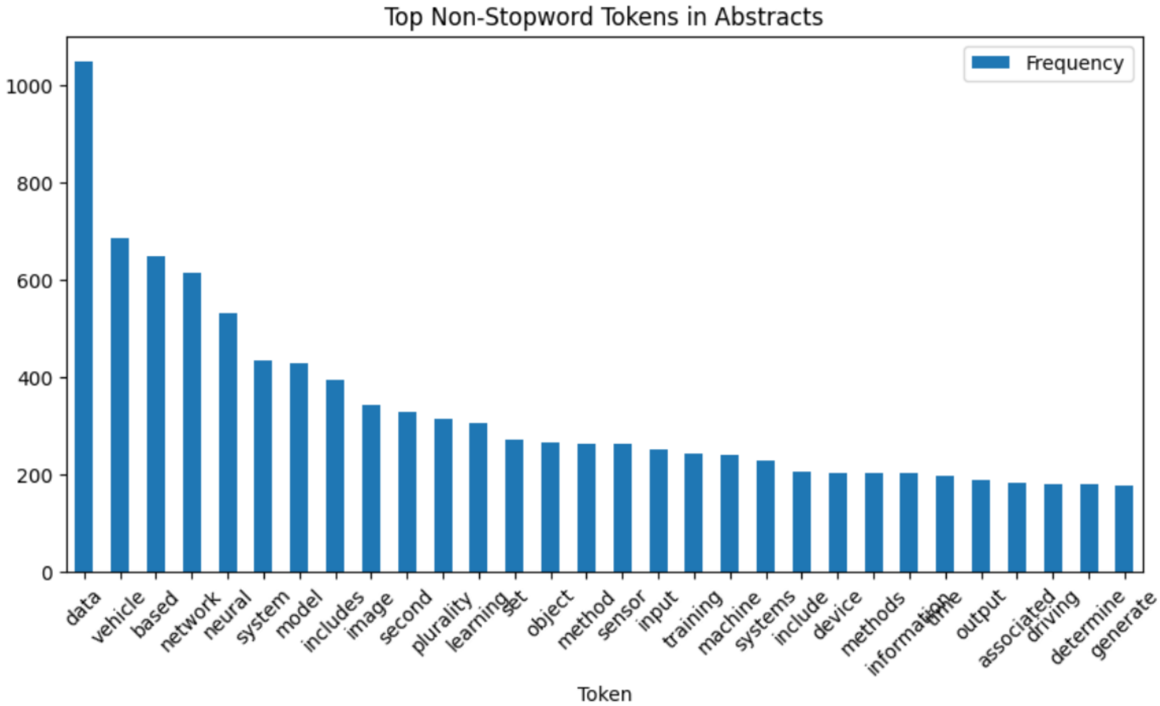



Top Applicants by Patent Count
INTEL CORP
NVIDIA CORP
FORD GLOBAL TECH LLC
QUALCOMM INC
TOYOTA RES INST INC
AMAZON TECH INC
BAIDU USA LLC
GOOGLE LLC
TOYOTA MOTOR CO LTD
APPLE INC
140
128
90
79
55
47
40
36
22
17
0
20
40
60
80
100
120
140
Number of Patents

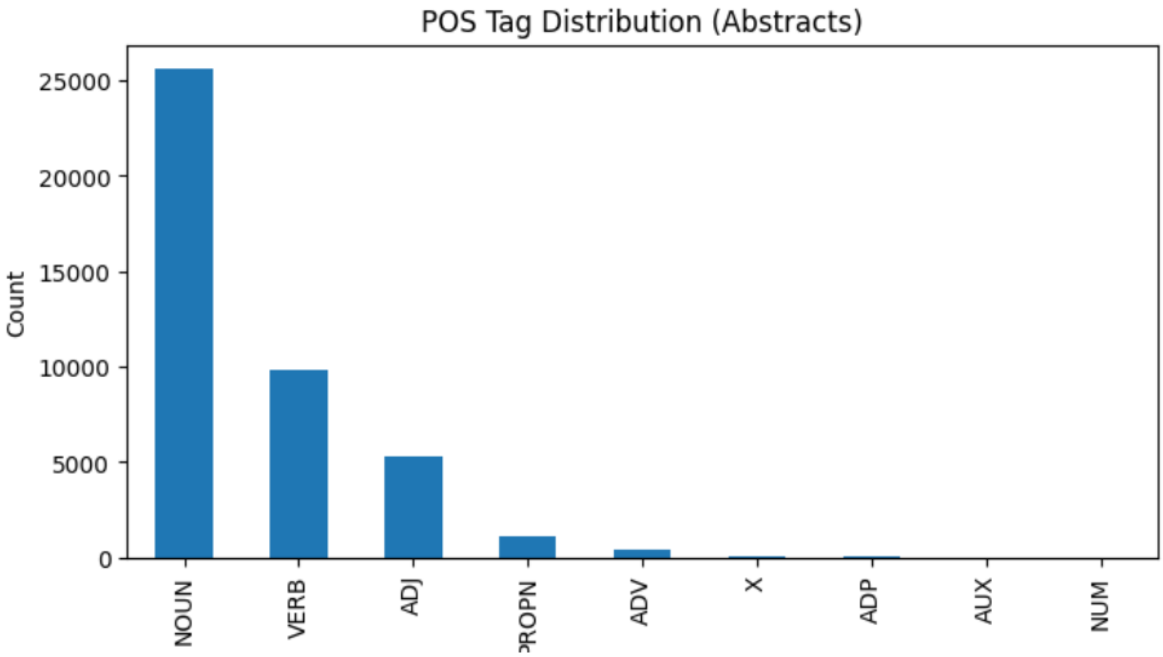

Top Inventors
Hu Jiangtao
Guim Bernat Francesc
Kautz Jan
Avadhanam Niranjan
Miao Jinghao
Barik Rajkishore
Chakravarty Punarjay
Chen Xiaoming
Cohen Taco Sebastiaan
Nister David
Koeberl Patrick
Fais Yaniv
Faivishevsky Lev
Fox Dieter
Luo Qi
Aila Timo Oskari
Dally William James
Diamant Ron
Jiang Shu
Jiang Yifei
Inventor Name Exact
0
2
4
6
8
10
12
14
16
18
20
Document Count


Top 15 CPC Codes
CPC Code
G06N3/045
G06N3/08
G06N20/00
G06N3/084
G06N3/044
G06N3/063
G06T2207/20084
G06N3/088
G06N3/04
G05D1/0088
G06N7/01
G06T2207/20081
G06N3/047
G06N3/048
G05D1/0221
Count

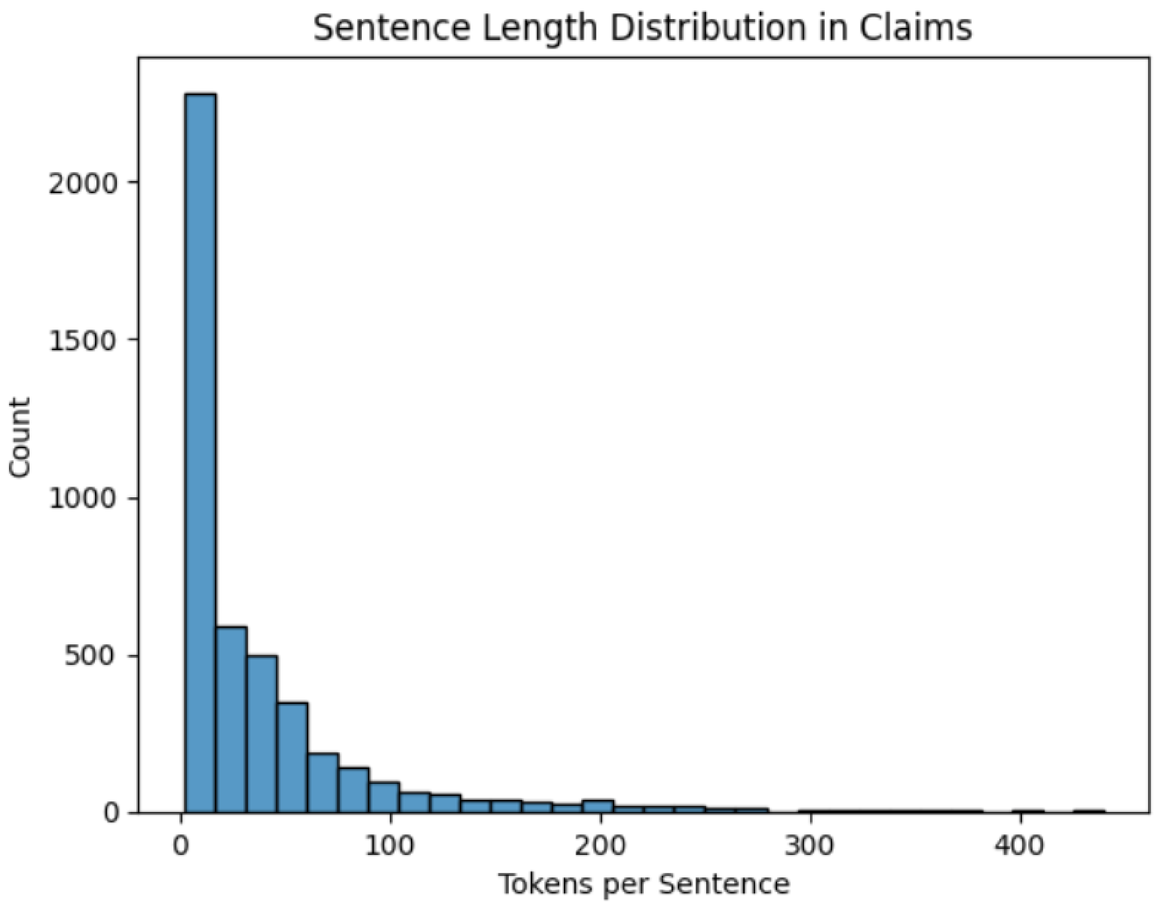

Distribution of Abstract Length
Distribution of Claims Length
Distribution of Description Length
Frequency
Word Count

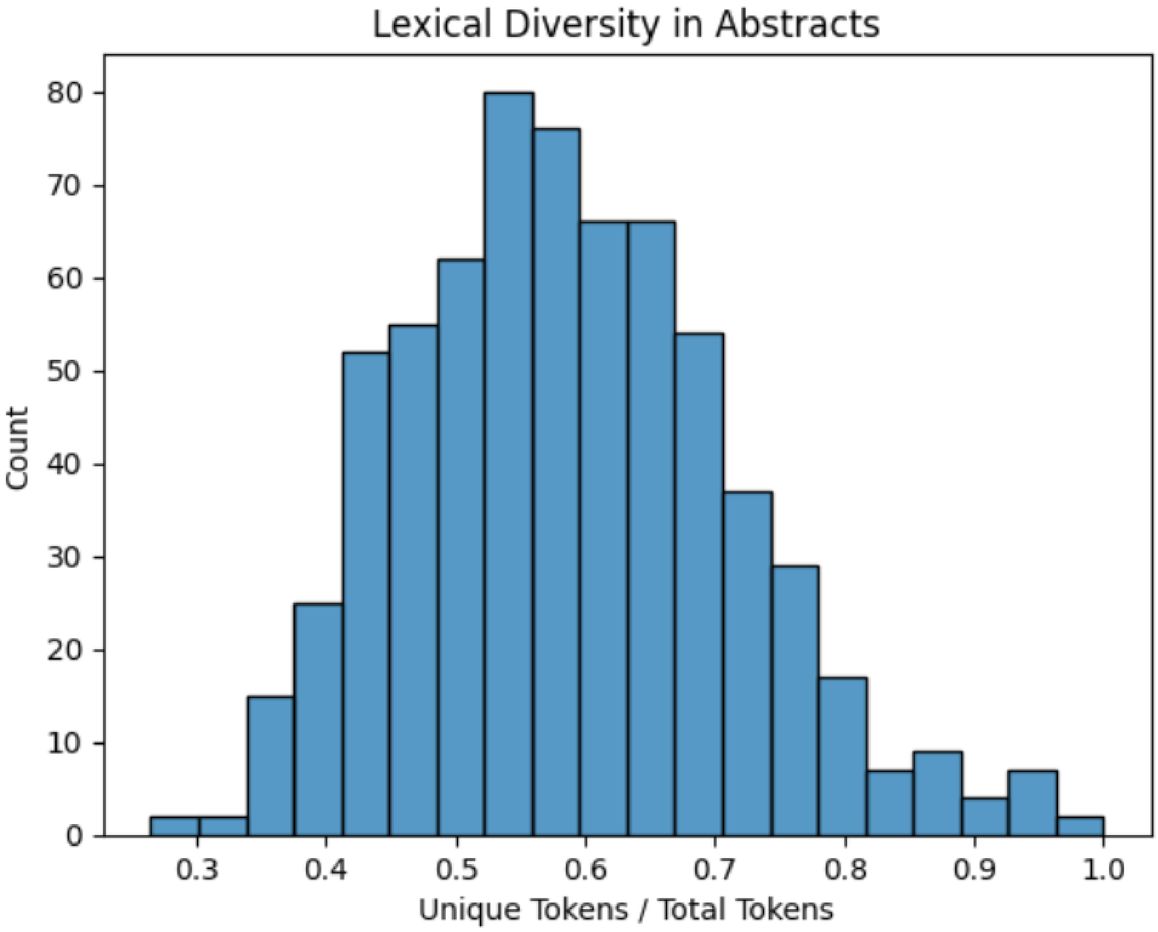


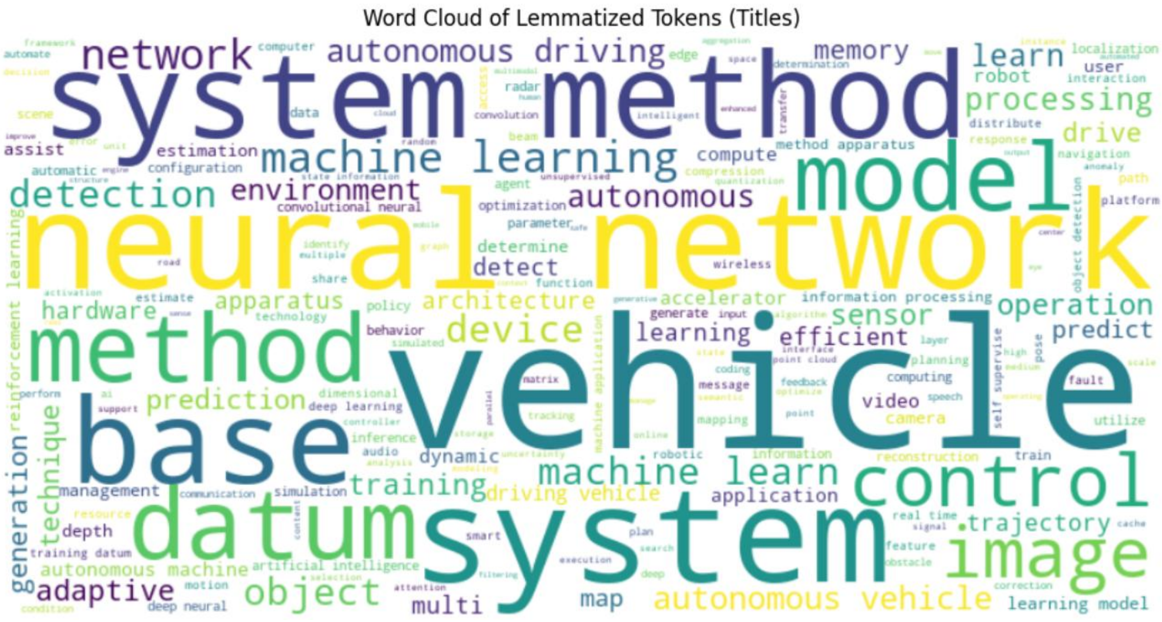
Top Non-Stopword Tokens in Abstracts
Frequency
1000
800
600
400
200
0
data
vehicle
based
network
neural
system
model
includes
image
second
plurality
learning
set
object
method
sensor
input
training
machine
systems
include
device
methods
information
time
output
associated
driving
determine
generate
Token



POS Tag Distribution (Abstracts)
Count
25000
20000
15000
10000
5000
0
NOUN
VERB
ADJ
PROPN
ADV
X
ADP
AUX
NUM

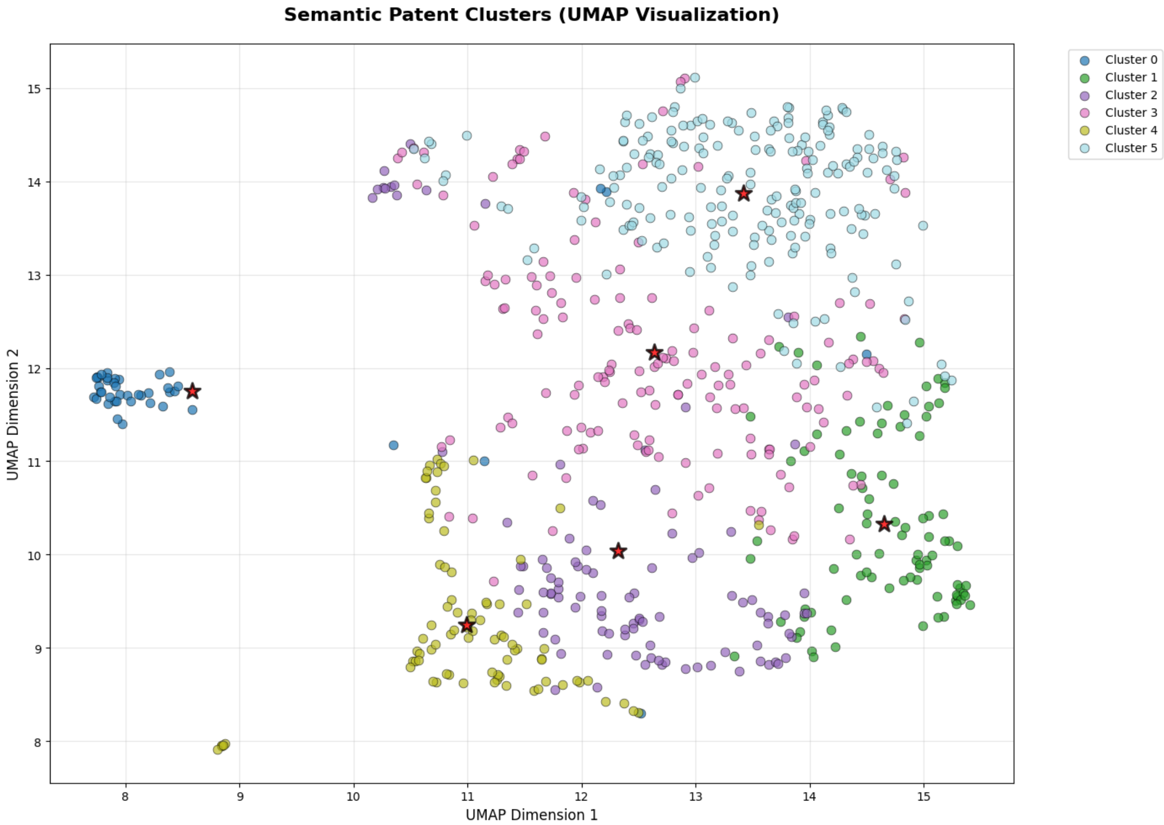

Sentence Length Distribution in Claims
Count
2000
1500
1000
500
0
0
100
200
300
400
Tokens per Sentence

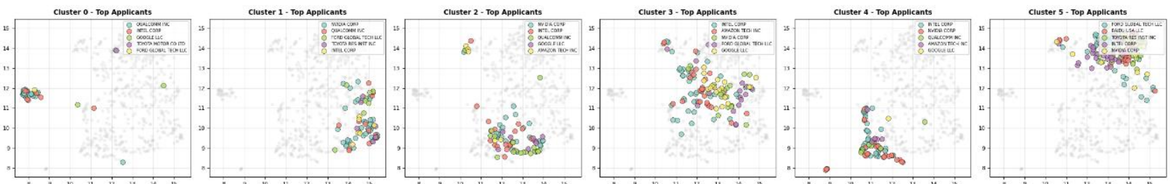

Lexical Diversity in Abstracts
Count
Unique Tokens / Total Tokens

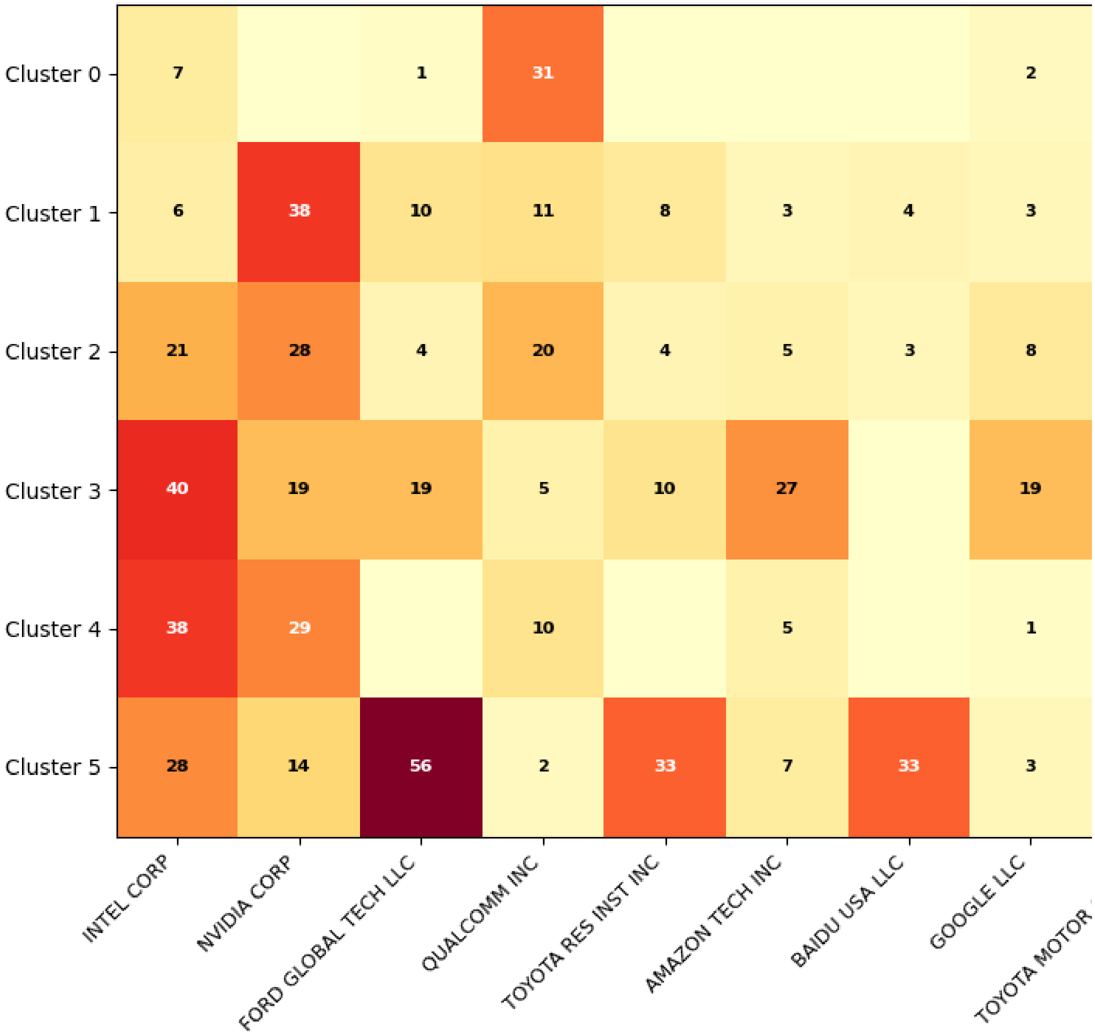

Word Cloud of Lemmatized Tokens (Titles)

Cluster ID	Technology Focus	Patents	Key Technologies	Semantic Coherence	Market Concentration
0	Wireless Communication	43	V2X, CSI, Wireless	0.47	High
1	Computer Vision	90	3D Vision, Depth, Camera	0.454	Medium
2	Neural Networks	99	Neural Arch, Deep Learning	0.435	Medium
3	AI/ML Applications	155	Data Models, Object Recognition	0.393	Low
4	Hardware Acceleration	83	Circuits, Processing, Memory	0.416	High
5	Vehicle Systems	197	Autonomous Driving, Sensors	0.47	Medium

Technology Cluster	Market Leader	Leader Share (%)	Patents	Key Competitors
Wireless Communication	QUALCOMM INC	72.1	31	Intel (16.3%), Google (4.7%)
Computer Vision	NVIDIA CORP	42.2	38	Qualcomm (12.2%), Ford (11.1%)
Neural Networks	NVIDIA CORP	28.3	28	Intel (21.2%), Qualcomm (20.2%)
AI/ML Applications	INTEL CORP	25.8	40	Amazon (17.4%), Nvidia (12.3%)
Hardware Acceleration	INTEL CORP	45.8	38	Nvidia (34.9%), Qualcomm (12.0%)
Vehicle Systems	FORD GLOBAL TECH LLC	28.4	56	Baidu USA (16.8%), Toyota Res (16.8%)

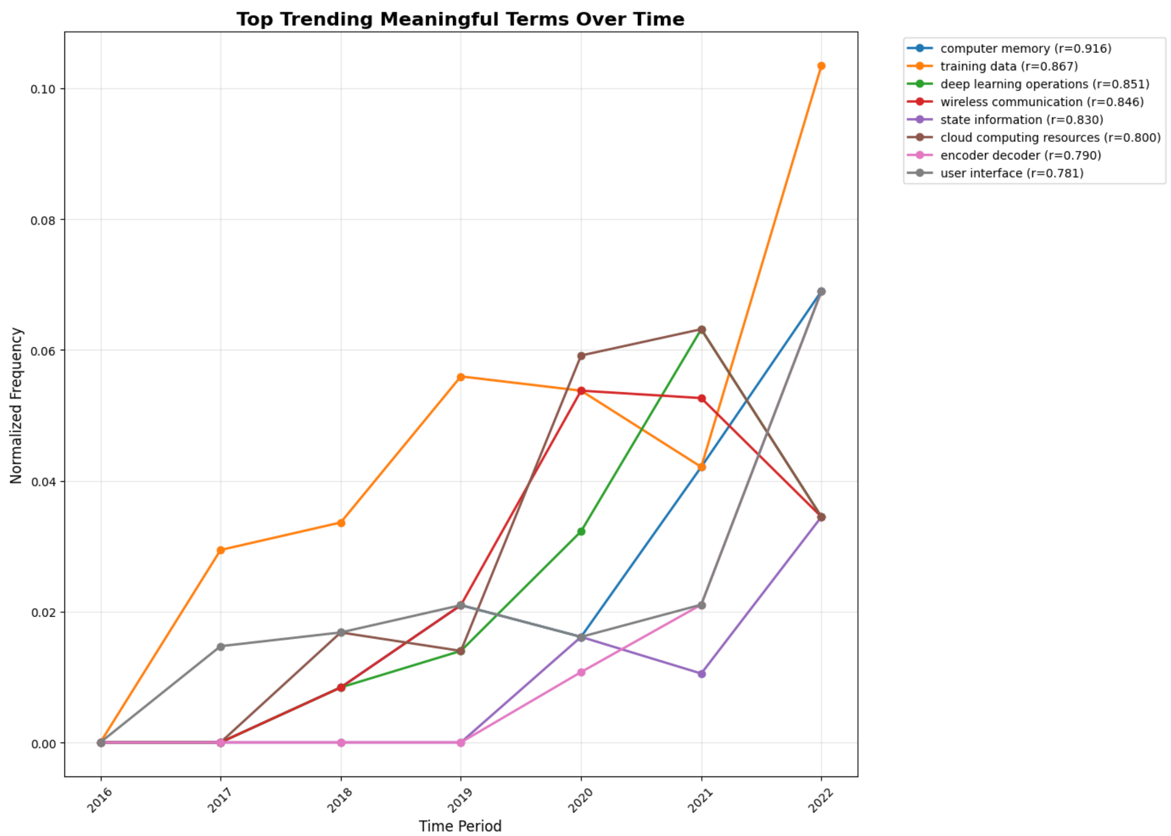

Semantic Patent Clusters (UMAP Visualization)
Cluster 0
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
UMAP Dimension 2
UMAP Dimension 1
8
9
10
11
12
13
14
15


Cluster 0 - Top Applicants
Cluster 1 - Top Applicants
Cluster 2 - Top Applicants
Cluster 3 - Top Applicants
Cluster 4 - Top Applicants
Cluster 5 - Top Applicants



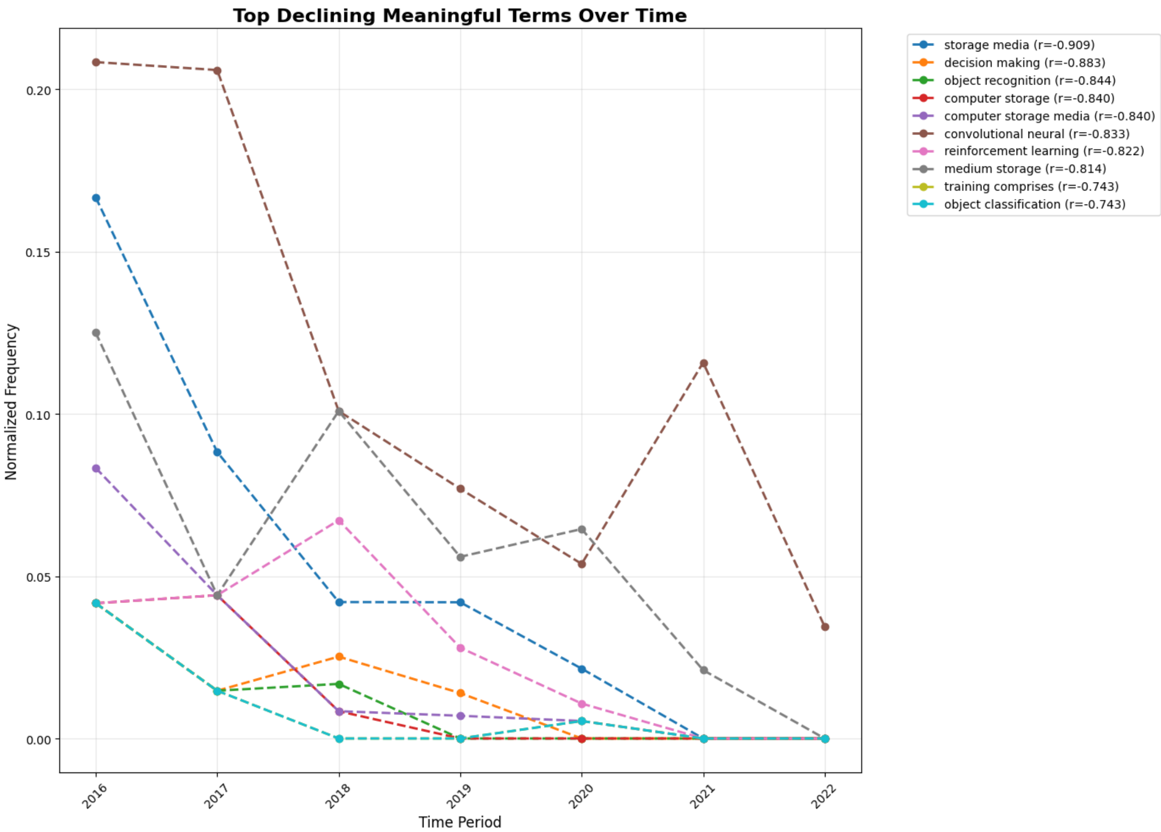
Metric	Value
Total Patents Analyzed	667
Technology Clusters Identified	6
Unique Applicant Companies	14
Innovation Timeline	2014-2022 (8+ years)
Most Active Cluster	Vehicle Systems (197 patents)
Highest Market Concentration	72.1% (Qualcomm in Wireless)


Innovation Trend Analysis Results
667
Total Patents
7
Years Analyzed
4,701
Unique Terms
9.29
Avg Terms/Patent


Top Trending Meaningful Terms Over Time
computer memory (r=0.916)
training data (r=0.867)
deep learning operations (r=0.851)
wireless communication (r=0.846)
state information (r=0.830)
cloud computing resources (r=0.800)
encoder decoder (r=0.790)
user interface (r=0.781)
Normalized Frequency
0.00
0.02
0.04
0.06
0.08
0.10
Time Period
2016
2017
2018
2019
2020
2021
2022

Top Emerging Technology Trends

Technology Term	Correlation (r)	Significance (p)	Trend
Computer Memory	0.916	0.004	↗ Strong Growth
Training Data	0.867	0.012	↗ Strong Growth
Deep Learning Operations	0.851	0.015	↗ Strong Growth
Wireless Communication	0.846	0.016	↗ Strong Growth
Cloud Computing Resources	0.800	0.031	↗ Growing



Top Declining Technology Trends

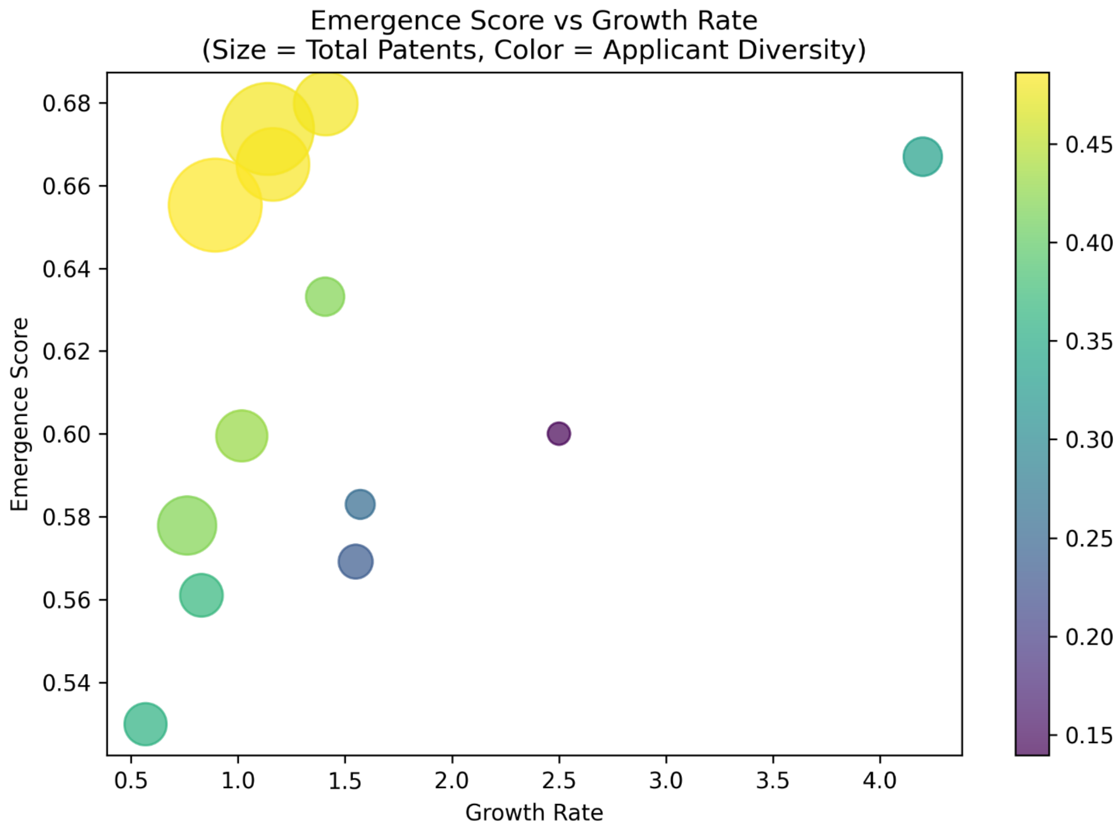
Technology Term	Correlation (r)	Significance (p)	Trend
Storage Media	-0.909	0.005	↘ Strong Decline
Decision Making	-0.883	0.008	↘ Strong Decline
Object Recognition	-0.844	0.017	↘ Strong Decline
Convolutional Neural	-0.833	0.020	↘ Declining
Reinforcement Learning	-0.822	0.023	↘ Declining

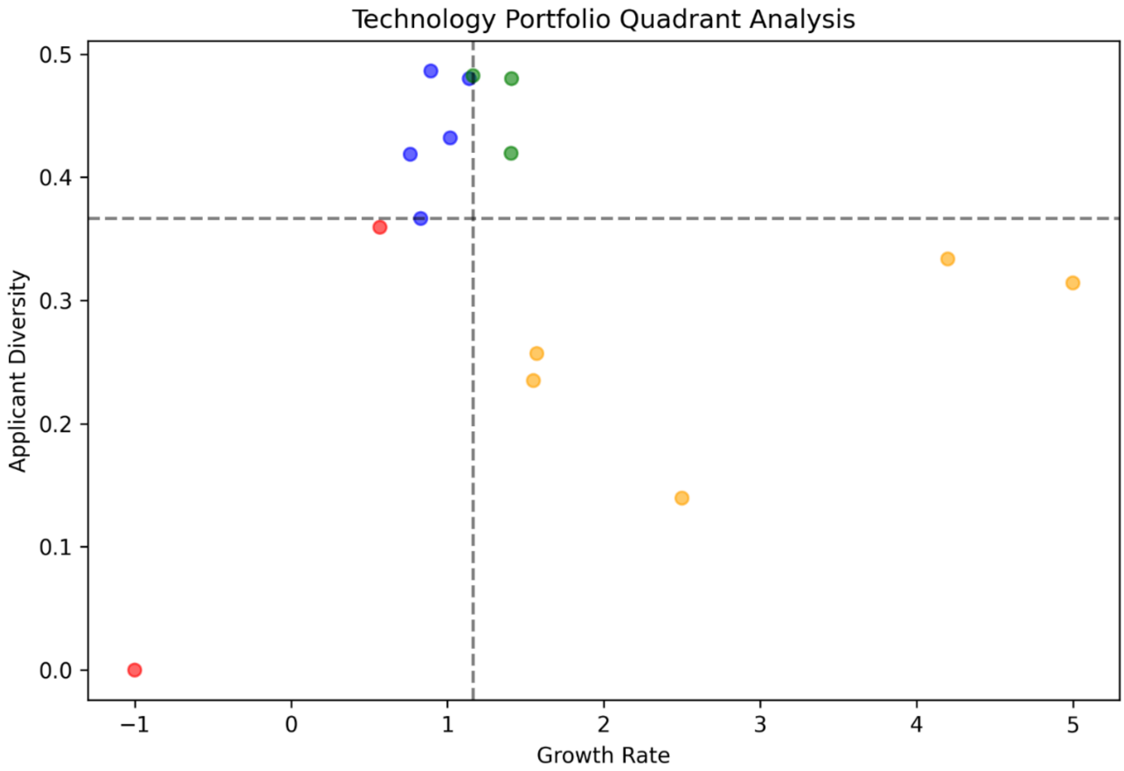
Leading Applicants & Focus Areas

Company	Patents	Primary Focus Areas
Intel Corp	140	Machine Learning, Collaborative Semantic Mapping, Memory Map
NVIDIA Corp	128	Encryption Standard, Operation Descriptor, Circuit ASIC
Ford Global Tech	90	Machine Learning, Computer Memory, Strain Displacement
Qualcomm Inc	79	Output Mask, Segmentation Neural, Coordinate Information
Toyota Research Institute	55	Odometry Noise Model, Motion Sensor, Fleet-scale Datasets

🎯 Top 5 Most Emerging AV Technologies

Rank	Technology	Emergence Score	Growth Rate	Strategic Significance
1	Perception & Sensing	0.680	1.41×	Highest emergence score - Critical for AV safety
2	Data Processing	0.674	1.14×	Second highest patents (563) + strong emergence
3	V2X Communication	0.667	4.20×	Breakthrough technology - Fastest established growth
4	Software Algorithms	0.665	1.16×	Third highest patents (349) + consistent emergence
5	AI/ML Architecture	0.655	0.89×	Most patented (574) but maturing - foundational tech





📈 Patent Volume vs. Emergence Comparison

Technology	Patent Count	Patent Rank	Emergence Rank	Gap Analysis
AI/ML Architecture	574	1	5	Mature technology - high volume, lower emergence
Data Processing	563	2	2	Balanced - high volume and emergence
Perception & Sensing	266	4	1	Emerging leader - lower volume, highest emergence
V2X Communication	94	9	3	Breakthrough - low volume, high emergence

Key Statistics

- **667** Total Patents Analyzed
- **100** Outliers Detected
- **15%** Outlier Rate
- **2024** Peak Innovation Year

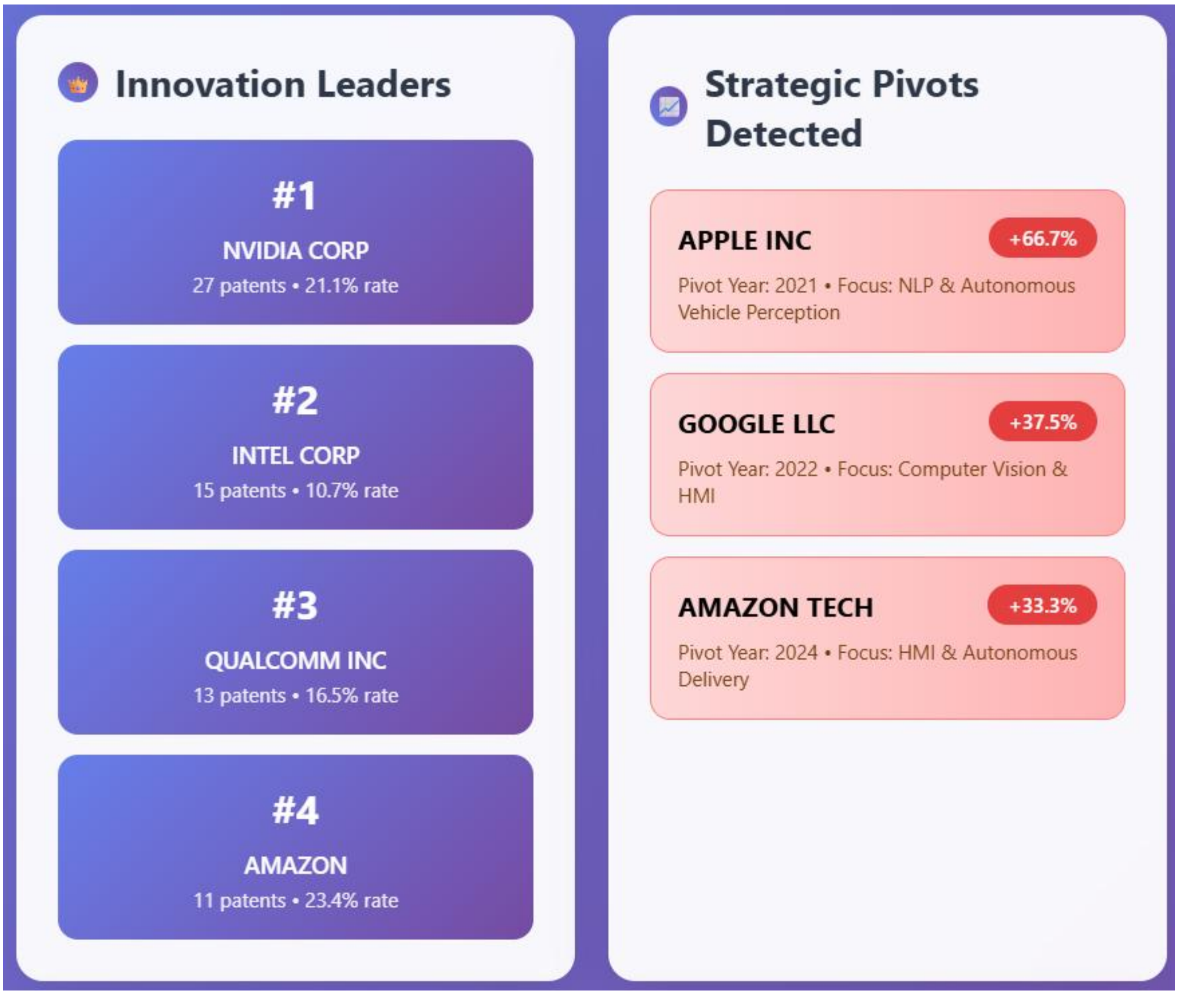
Dominant Technology Areas

- AI/ML Architecture
- Computer Vision
- Human-Machine Interface
- Autonomous Vehicle Perception
- Video Compression
- Wireless Communication
- Sensor Fusion
- Circuit Simulation
- Autonomous Delivery

Key Insight: QUALCOMM dominates top innovations with diverse technology portfolio, while NVIDIA leads in volume. Major tech giants are pivoting strategies, with Apple showing the most dramatic 66.7% increase in radical innovation rate since 2021.

Top 5 Radical Innovations (Highest Outlier Scores)

1	Video compression using recurrent-based machine learning systems	QUALCOMM INC	Video Compression Technology	0.767
2	Dynamic uplink data split threshold	QUALCOMM INC	Wireless Communication	0.696
3	Radar-aided single image three-dimensional depth reconstruction	QUALCOMM INC	Sensor Fusion for Depth Reconstruction	0.628
4	Locally and globally locating actors by digital cameras	AMAZON TECH INC	Computer Vision for Autonomous Vehicles	0.613
5	Layout parasitics and device parameter prediction using graphs	NVIDIA CORP	Circuit Simulation and Design	0.576



10.2 PLAGIARISM REPORT

11. REFERENCES

- **Base Paper:**

  - Trappey, A. J. C., Trappey, C. V., Wu, J.-L., & Wang, J. W. C. (2020). Intelligent compilation of patent summaries using machine learning and natural language processing techniques. *Advanced Engineering Informatics, 43*, 101027. https://doi.org/10.1016/j.aei.2019.101027

- **Paper I:**

- Cho, R. L. T., Liu, J. S., & Ho, M. H. C. (2021). The development of autonomous driving technology: perspectives from patent citation analysis. *Transport Reviews*, *41*(5), 685–711. **https://doi.org/10.1080/01441647.2021.1879310**

- **Paper II:**
    - Kim, G., & Bae, J. (2017). A novel approach to forecast promising technology through patent analysis. *Technological Forecasting and Social Change, 117*, 228–237. **https://doi.org/10.1016/j.techfore.2016.11.023**

- **Paper III:**
    - Lattimer, B., Chen, P. H., Zhang, X., & Yang, Y. (2023). Fast and accurate factual inconsistency detection over long documents. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 1691-1703). Association for Computational Linguistics. **https://doi.org/10.18653/v1/2023.emnlp-main.105**