Universidad de Monterrey

Integración de aplicaciones computacionales

Ejercicio Guiado 1

Prompt Engineering con Cursor AI

Maestro: Dr. Raúl Morales Salcedo

Nombre: Margarita Concepción Cuervo Citalán #581771

Carrera: ITC, 9no Semestre

Día y hora del grupo de la clase: lunes y jueves 10:00h

7 de agosto de 2025.

Doy mi palabra de haber realizado esta actividad con integridad académica.

ÍNDICE

Introducción

Este ejercicio guiado tiene como objetivo desarrollar una aplicación CLI en Python que permita clasificar servicios en la nube según su modelo (IaaS, PaaS, SaaS, FaaS). A través del uso de reglas básicas y técnicas de inteligencia artificial, se busca reforzar habilidades en programación, validación de datos y manejo de argumentos en línea de comandos, complementado con la creación de un sitio web para documentar y reflexionar sobre el proceso.

Desarrollo

Se empezó por usar la inteligencia artificial que cuenta el IDE de Cursor para este ejercicio donde el primer prompt que se le introdujo fue: *Genera un código Python que reciba un texto y clasifique si corresponde a IaaS, PaaS o FaaS usando reglas básicas.* Tras escribir eso la IA arrojó un archivo:

1. **cloud_classifier.py:** contiene las funciones para determinar que tipo de modelo se está hablando por medio de funciones que contienen un diccionario de palabras y patrones clave para cada tipo de modelo de nube, patrones regex para identificar patrones más complejos, un clasificador de texto y un generador de puntuaciones para que con base en ello determine si si habla de algún modelo en concreto o no se sabe. Asimismo se prueba este primer intento con textos ya predefinidos.

El código de **cloud_classifier.py** es el siguiente:

```python
import re
from typing import Dict, List, Tuple

class CloudModelClassifier:
    """
    Clasificador de modelos de nube basado en reglas para identificar
    si un texto corresponde a IaaS, PaaS o FaaS
    """

    def __init__(self):
        # Palabras clave y patrones para cada modelo de nube
        self.iaas_keywords = [
            'infraestructura', 'infrastructure', 'servidor', 'server', 'vm', 'virtual
machine',
```

```
            'almacenamiento', 'storage', 'red', 'network', 'computo', 'compute',
'hardware',
            'provisionamiento', 'provisioning', 'escalabilidad', 'scalability',
'monitoreo',
            'monitoring', 'backup', 'disaster recovery', 'load balancer', 'firewall',
            'vpc', 'virtual private cloud', 'subnet', 'gateway', 'elastic ip'
        ]

        self.paas_keywords = [
            'plataforma', 'platform', 'desarrollo', 'development', 'deployment',
'despliegue',
            'runtime', 'entorno de ejecucion', 'middleware', 'database service',
'servicio de base de datos',
            'authentication', 'autenticacion', 'api management', 'gestion de apis',
'ci/cd',
            'continuous integration', 'continuous deployment', 'build tools',
'herramientas de construccion',
            'container orchestration', 'orquestacion de contenedores', 'kubernetes',
'docker swarm'
        ]

        self.faas_keywords = [
            'funcion', 'function', 'serverless', 'sin servidor', 'event-driven',
'dirigido por eventos',
            'lambda', 'azure functions', 'google cloud functions', 'openwhisk',
'knative',
            'microservicios', 'microservices', 'api gateway', 'triggers',
'disparadores',
            'cold start', 'warm start', 'execution time', 'tiempo de ejecucion',
'stateless',
            'sin estado', 'ephemeral', 'efimero', 'auto-scaling', 'auto-escalado'
        ]

        # Patrones regex para identificar conceptos más complejos
        self.patterns = {
            'iaas': [
                r'\b(ec2|rds|s3|vpc|ecs|eks|lambda)\b',  # AWS services
                r'\b(vm|virtual machine|instance)\b',
                r'\b(storage|block storage|object storage)\b',
                r'\b(network|subnet|gateway|router)\b'
            ],
            'paas': [
                r'\b(elastic beanstalk|app engine|heroku|openshift)\b',
                r'\b(deploy|deployment|build|compile)\b',
                r'\b(database|redis|mongodb|postgresql)\b',
                r'\b(authentication|oauth|jwt)\b'
            ],
            'faas': [
                r'\b(aws lambda|azure functions|google cloud functions)\b',
                r'\b(serverless|function as a service)\b',
                r'\b(event|trigger|webhook)\b',
                r'\b(cold start|warm start)\b'
            ]
        }
```

```
def classify_text(self, text: str) -> Dict[str, any]:
    """
    Clasifica el texto según el modelo de nube que mejor se ajuste

    Args:
        text (str): Texto a clasificar

    Returns:
        Dict con la clasificación y puntuaciones
    """
    if not text or not text.strip():
        return {
            'classification': 'unknown',
            'confidence': 0.0,
            'scores': {'iaas': 0, 'paas': 0, 'faas': 0},
            'reasoning': 'Texto vacío o nulo'
        }

    # Convertir a minúsculas para comparación
    text_lower = text.lower()

    # Calcular puntuaciones basadas en palabras clave
    scores = {
        'iaas': self._calculate_keyword_score(text_lower, self.iaas_keywords),
        'paas': self._calculate_keyword_score(text_lower, self.paas_keywords),
        'faas': self._calculate_keyword_score(text_lower, self.faas_keywords)
    }

    # Aplicar patrones regex para puntuaciones adicionales
    scores = self._apply_pattern_scores(text_lower, scores)

    # Normalizar puntuaciones
    total_score = sum(scores.values())
    if total_score > 0:
        scores = {k: v/total_score for k, v in scores.items()}

    # Determinar la clasificación
    classification = max(scores, key=scores.get)
    confidence = scores[classification]

    # Generar razonamiento
    reasoning = self._generate_reasoning(text_lower, scores, classification)

    return {
        'classification': classification.upper(),
        'confidence': round(confidence, 3),
        'scores': {k: round(v, 3) for k, v in scores.items()},
        'reasoning': reasoning
    }

def _calculate_keyword_score(self, text: str, keywords: List[str]) -> float:
    """Calcula puntuación basada en palabras clave"""
    score = 0.0
```

```
        for keyword in keywords:
            if keyword in text:
                score += 1.0
                # Bonus por palabras clave más específicas
                if len(keyword) > 8:
                    score += 0.5
        return score

    def _apply_pattern_scores(self, text: str, scores: Dict[str, float]) -> Dict[str,
float]:
        """Aplica puntuaciones adicionales basadas en patrones regex"""
        for model, patterns in self.patterns.items():
            for pattern in patterns:
                matches = re.findall(pattern, text)
                if matches:
                    scores[model] += len(matches) * 0.5
        return scores

    def _generate_reasoning(self, text: str, scores: Dict[str, float], classification:
str) -> str:
        """Genera explicación del razonamiento de clasificación"""
        reasons = []

        # Agregar razones basadas en puntuaciones
        for model, score in scores.items():
            if score > 0:
                reasons.append(f"{model.upper()}: {score:.3f}")

        # Agregar razones específicas
        if classification == 'iaas':
            if any(word in text for word in ['servidor', 'vm', 'hardware']):
                reasons.append("Menciona infraestructura física/virtual")
        elif classification == 'paas':
            if any(word in text for word in ['desarrollo', 'deploy', 'plataforma']):
                reasons.append("Enfocado en desarrollo y despliegue")
        elif classification == 'faas':
            if any(word in text for word in ['funcion', 'serverless', 'lambda']):
                reasons.append("Menciona funciones sin servidor")

        return f"Clasificado como {classification.upper()} basado en: {',
'.join(reasons)}"

    def classify_multiple_texts(self, texts: List[str]) -> List[Dict[str, any]]:
        """
        Clasifica múltiples textos

        Args:
            texts (List[str]): Lista de textos a clasificar

        Returns:
            Lista de clasificaciones
        """
        return [self.classify_text(text) for text in texts]
```

```
    def get_classification_summary(self, texts: List[str]) -> Dict[str, any]:
        """
        Genera un resumen de clasificaciones para múltiples textos

        Args:
            texts (List[str]): Lista de textos a clasificar

        Returns:
            Resumen de clasificaciones
        """
        classifications = self.classify_multiple_texts(texts)

        # Contar clasificaciones
        counts = {'IaaS': 0, 'PaaS': 0, 'FaaS': 0, 'UNKNOWN': 0}
        total_confidence = 0

        for result in classifications:
            counts[result['classification']] += 1
            total_confidence += result['confidence']

        avg_confidence = total_confidence / len(classifications) if classifications
else 0

        return {
            'total_texts': len(texts),
            'classification_counts': counts,
            'average_confidence': round(avg_confidence, 3),
            'detailed_results': classifications
        }


def main():
    """Función principal para demostrar el uso del clasificador"""

    # Crear instancia del clasificador
    classifier = CloudModelClassifier()

    # Ejemplos de texto para clasificar
    example_texts = [
        "AWS EC2 proporciona instancias de servidores virtuales con almacenamiento EBS
y permite escalabilidad automática",
        "Heroku es una plataforma que facilita el despliegue de aplicaciones web con
herramientas de CI/CD integradas",
        "Azure Functions permite ejecutar código sin servidor basado en eventos y
triggers",
        "Google Cloud Platform ofrece servicios de infraestructura como servidores,
redes y almacenamiento",
        "Kubernetes es una plataforma de orquestación de contenedores para desplegar
aplicaciones",
        "AWS Lambda ejecuta funciones sin servidor que se activan por eventos
específicos"
    ]

    print("=== CLASIFICADOR DE MODELOS DE NUBE ===\n")
```

```
    # Clasificar cada texto individualmente
    for i, text in enumerate(example_texts, 1):
        print(f"Texto {i}: {text}")
        result = classifier.classify_text(text)
        print(f"Clasificación: {result['classification']}")
        print(f"Confianza: {result['confidence']}")
        print(f"Puntuaciones: {result['scores']}")
        print(f"Razonamiento: {result['reasoning']}")
        print("-" * 80)

    # Generar resumen general
    print("\n=== RESUMEN GENERAL ===")
    summary = classifier.get_classification_summary(example_texts)
    print(f"Total de textos: {summary['total_texts']}")
    print(f"Conteo de clasificaciones: {summary['classification_counts']}")
    print(f"Confianza promedio: {summary['average_confidence']}")


if __name__ == "__main__":
    main()
```

Dado que este primer código aun le faltan diversas mejoras, se le pedirán 3 mejoras: que añade una validación de entrada, que use una función para cada clasificación y probar este código con 5 ejemplos distintos.

Tras ejecutar las instrucciones, la IA cambió el diseño del código y ahora implementa las funciones de **validate_input**, que valida el texto de entrada, y una función por cada tipo de modelo, IaaS, SaaS, PaaS y FaaS. Asimismo, si no sabe identificar de qué modelo se habla, todos los puntajes se quedan en cero y no ofrece una respuesta.

Esto se debe a que en definiciones más complejas fuera de las básicas como por ejemplo: *tiene todos los lenguajes y herramientas para poder programar un sistema*, que se refiere a un PaaS, el código ya no funciona, por lo que se le pedirá ayuda a chat GPT para mejorar el código al integrar mejores validaciones de entrada y argparse.

Tras probarlo con definiciones menos convencionales, este clasificador arroja resultados bastante satisfactorios por lo que el código final es el siguiente:

Python

```
# cloud_classifier.py
import re
import argparse
import unicodedata
from typing import Dict, List, Tuple
from difflib import SequenceMatcher
from collections import defaultdict

class CloudModelClassifier:
```

```
    """
    Clasificador de modelos de nube (IaaS, PaaS, FaaS, SaaS).
    Mejoras:
      - Normalización y tokenización robusta
      - Fuzzy matching para palabras no exactas (difflib)
      - Similitud semántica contra prototipos por clase
      - No falla con palabras desconocidas; genera razonamiento transparente
    """
    def __init__(self):
        self.iaas_keywords = [
            'infraestructura', 'infrastructure', 'servidor', 'server', 'vm', 'virtual
machine',
            'almacenamiento', 'storage', 'red', 'network', 'computo', 'compute',
'hardware',
            'provisionamiento', 'provisioning', 'escalabilidad', 'scalability',
'monitoreo',
            'monitoring', 'backup', 'disaster recovery', 'load balancer', 'firewall',
            'vpc', 'virtual private cloud', 'subnet', 'gateway', 'elastic ip'
        ]
        self.paas_keywords = [
            'plataforma', 'platform', 'desarrollo', 'development', 'deployment',
'despliegue',
            'runtime', 'entorno de ejecucion', 'middleware', 'database service',
'servicio de base de datos',
            'authentication', 'autenticacion', 'api management', 'gestion de apis',
'ci/cd',
            'continuous integration', 'continuous deployment', 'build tools',
'herramientas de construccion',
            'container orchestration', 'orquestacion de contenedores', 'kubernetes',
'docker swarm',
            'lenguajes', 'frameworks', 'framework'
        ]
        self.faas_keywords = [
            'funcion', 'function', 'serverless', 'sin servidor', 'event-driven',
'dirigido por eventos',
            'lambda', 'azure functions', 'google cloud functions', 'openwhisk',
'knative',
            'microservicios', 'microservices', 'api gateway', 'triggers',
'disparadores',
            'cold start', 'warm start', 'execution time', 'tiempo de ejecucion',
'stateless',
            'sin estado', 'ephemeral', 'efimero', 'auto-scaling', 'auto-escalado'
        ]
        self.saas_keywords = [
            'software', 'aplicacion', 'application', 'servicio', 'service', 'usuario
final', 'end user',
            'subscription', 'suscripcion', 'licencia', 'license', 'web app',
'aplicacion web',
            'mobile app', 'aplicacion movil', 'dashboard', 'panel de control',
'reportes', 'reports',
            'analytics', 'analitica', 'crm', 'erp', 'hr software', 'software de
recursos humanos',
            'accounting software', 'software de contabilidad', 'collaboration',
'colaboracion',
```

```
            'productivity tools', 'herramientas de productividad', 'office suite',
'suite de oficina'
        ]

        self.patterns = {
            'iaas': [
                r'\b(ec2|rds|s3|vpc|ecs|eks)\b',
                r'\b(vm|virtual machine|instance)\b',
                r'\b(storage|block storage|object storage)\b',
                r'\b(network|subnet|gateway|router)\b'
            ],
            'paas': [
                r'\b(elastic beanstalk|app engine|heroku|openshift)\b',
                r'\b(deploy|deployment|build|compile|runtime)\b',
                r'\b(database|redis|mongodb|postgresql)\b',
                r'\b(authentication|oauth|jwt)\b'
            ],
            'faas': [
                r'\b(aws lambda|azure functions|google cloud
functions|openwhisk|knative)\b',
                r'\b(serverless|function as a service|function)\b',
                r'\b(event|trigger|webhook)\b',
                r'\b(cold start|warm start)\b'
            ],
            'saas': [
                r'\b(salesforce|office 365|google workspace|slack|zoom|dropbox)\b',
                r'\b(web application|web app|mobile application|mobile app)\b',
                r'\b(subscription|suscripcion|licencia|license)\b',
                r'\b(user interface|interfaz de usuario|dashboard|panel)\b'
            ]
        }

        # Prototipos que sirven para una similitud semántica muy básica
        self.prototypes = {
            'iaas': "servidores virtuales infraestructura almacenamiento redes
provisionamiento hardware",
            'paas': "plataforma desarrollo despliegue runtimes lenguajes frameworks
herramientas ci cd",
            'faas': "funciones sin servidor eventos triggers ejecucion on demand
lambda serverless",
            'saas': "aplicacion lista usuario final suscripcion software servicio
dashboard colaboracion"
        }
        self.prototype_tokens = {k: set(self._tokenize(self._normalize(v))) for k, v
in self.prototypes.items()}

    # ---------- utilidades ----------
    def _normalize(self, text: str) -> str:
        if text is None:
            return ""
        text = text.lower()
        text = unicodedata.normalize('NFKD', text)
        text = "".join([c for c in text if not unicodedata.combining(c)])
        text = re.sub(r'[^a-z0-9\s]', ' ', text)
```

```
        return re.sub(r'\s+', ' ', text).strip()

    def _tokenize(self, text: str) -> List[str]:
        return [t for t in text.split() if len(t) > 1]

    def _best_fuzzy_ratio(self, token: str, candidates: List[str]) -> float:
        best = 0.0
        for c in candidates:
            r = SequenceMatcher(None, token, c).ratio()
            if r > best:
                best = r
                if best >= 0.99:
                    break
        return best

    # ---------- scoring ----------
    def _calculate_keyword_score(self, text: str, keywords: List[str]) -> Tuple[float,
List[Tuple[str, float]]]:
        score = 0.0
        details = []
        norm = self._normalize(text)
        tokens = self._tokenize(norm)
        for kw in keywords:
            kw_norm = self._normalize(kw)
            if kw_norm in norm:
                score += 1.0
                details.append((kw, 1.0))
                continue
            best_ratio = self._best_fuzzy_ratio(kw_norm, tokens)
            if best_ratio >= 0.85:
                score += 0.9 * best_ratio
                details.append((kw, best_ratio))
            elif best_ratio >= 0.6:
                score += 0.5 * best_ratio
                details.append((kw, best_ratio))
        return score, details

    def _apply_pattern_score(self, text: str, model: str) -> Tuple[float, List[str]]:
        score = 0.0
        matches = []
        norm = self._normalize(text)
        if model in self.patterns:
            for pattern in self.patterns[model]:
                found = re.findall(pattern, norm)
                if found:
                    increment = len(found) * 0.7
                    score += increment
                    matches.append(pattern)
        return score, matches

    def _semantic_score(self, text: str, model: str) -> float:
        norm = self._normalize(text)
        tokens = set(self._tokenize(norm))
        proto = self.prototype_tokens.get(model, set())
```

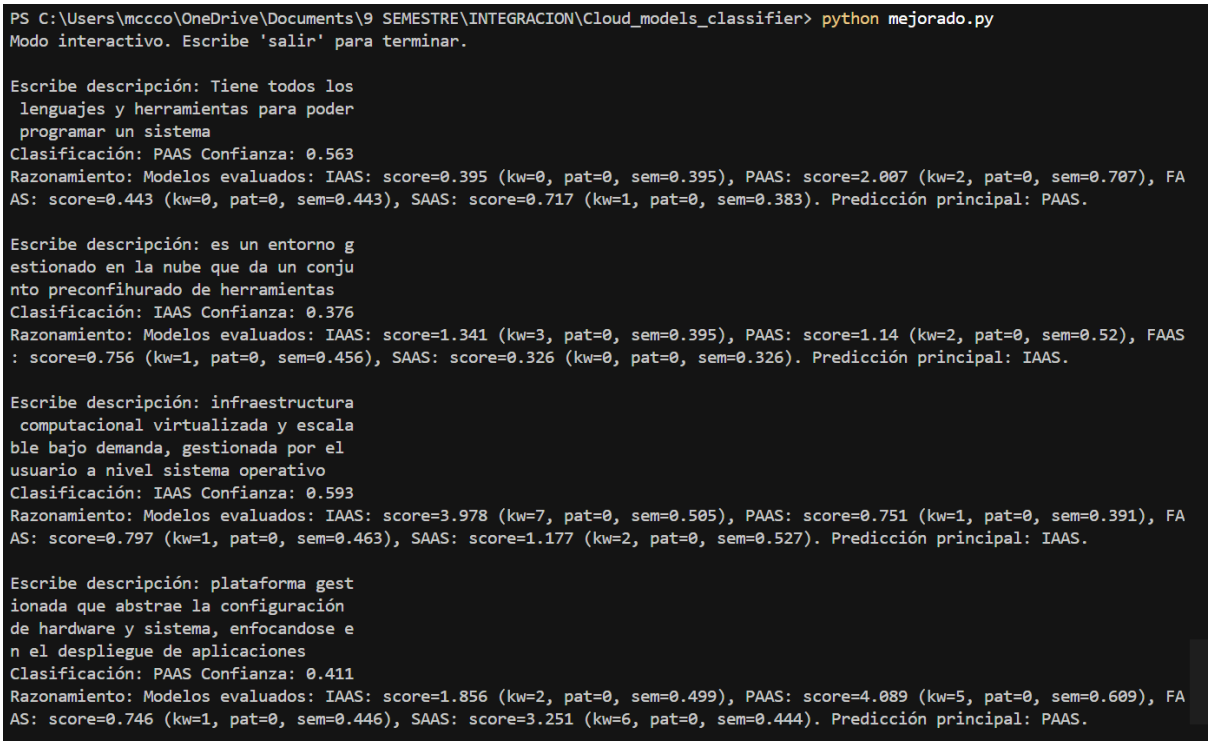
```
        if not tokens or not proto:
            return 0.0
        intersection = tokens.intersection(proto)
        union = tokens.union(proto)
        jaccard = len(intersection) / len(union) if union else 0.0
        total_ratios = 0.0
        count = 0
        for t in tokens:
            best = self._best_fuzzy_ratio(t, list(proto))
            total_ratios += best
            count += 1
        avg_ratio = (total_ratios / count) if count else 0.0
        semantic = jaccard * 1.5 + avg_ratio * 1.0
        return semantic

    # ---------- razonamiento ----------
    def _generate_reasoning(self, text: str, per_model_raw: Dict[str, Dict]) -> str:
        parts = []
        for model, info in per_model_raw.items():
            score = info['raw_score']
            kws = info.get('keyword_matches', [])
            pats = info.get('pattern_matches', [])
            sem = round(info.get('semantic_score', 0.0), 3)
            parts.append(f"{model.upper()}: score={round(score,3)} (kw={len(kws)},
pat={len(pats)}, sem={sem})")
        ranked = sorted(per_model_raw.items(), key=lambda x: x[1]['raw_score'],
reverse=True)
        top = ranked[0][0].upper() if ranked else 'N/A'
        return f"Modelos evaluados: {', '.join(parts)}. Predicción principal: {top}."

    # ---------- API pública ----------
    def classify_text(self, text: str) -> Dict[str, any]:
        try:
            if not text or not isinstance(text, str) or len(text.strip()) < 3:
                return {
                    'classification': 'ERROR',
                    'confidence': 0.0,
                    'scores': {},
                    'reasoning': 'Texto inválido o demasiado corto (mínimo 3
caracteres)'
                }

            per_model_raw = {}
            models = {
                'iaas': (self.iaas_keywords, 'iaas'),
                'paas': (self.paas_keywords, 'paas'),
                'faas': (self.faas_keywords, 'faas'),
                'saas': (self.saas_keywords, 'saas'),
            }
            for m, (kw_list, _) in models.items():
                kw_score, kw_matches = self._calculate_keyword_score(text, kw_list)
                pat_score, pat_matches = self._apply_pattern_score(text, m)
                sem_score = self._semantic_score(text, m)
                raw_score = kw_score + pat_score + sem_score
```

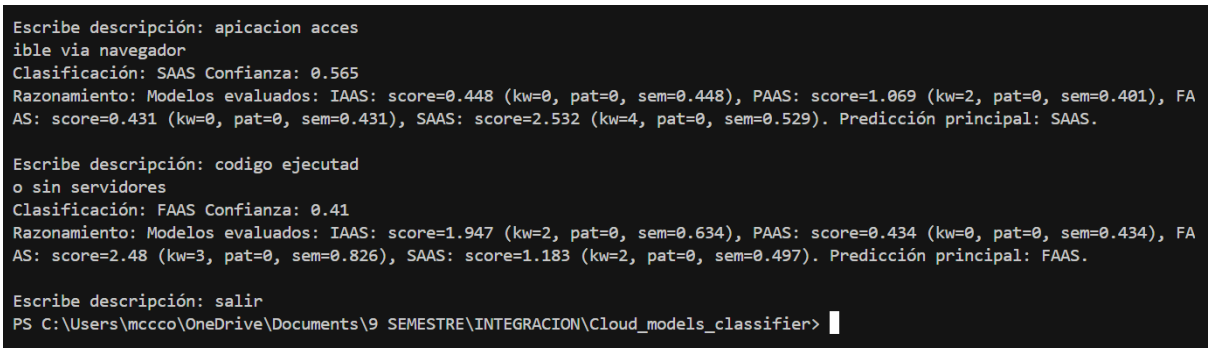
```
                per_model_raw[m] = {
                    'keyword_score': round(kw_score, 3),
                    'keyword_matches': kw_matches,
                    'pattern_score': round(pat_score,3),
                    'pattern_matches': pat_matches,
                    'semantic_score': round(sem_score,3),
                    'raw_score': round(raw_score, 3)
                }

            total_raw = sum(info['raw_score'] for info in per_model_raw.values())
            if total_raw <= 0:
                return {
                    'classification': 'UNKNOWN',
                    'confidence': 0.0,
                    'scores': {k: 0.0 for k in per_model_raw.keys()},
                    'reasoning': "No se hallaron coincidencias relevantes en palabras
clave, patrones o prototipos."
                }

            normalized = {k: v['raw_score']/total_raw for k, v in
per_model_raw.items()}
            classification = max(normalized, key=normalized.get)
            confidence = normalized[classification]
            final_class = 'UNKNOWN' if confidence < 0.20 else classification.upper()
                reasoning = self._generate_reasoning(text, per_model_raw)

            return {
                'classification': final_class,
                'confidence': round(confidence, 3),
                'scores': {k: round(v,3) for k, v in normalized.items()},
                'details': per_model_raw,
                'reasoning': reasoning
            }
        except Exception as e:
            return {
                'classification': 'ERROR',
                'confidence': 0.0,
                'scores': {},
                'reasoning': f'Error inesperado durante la clasificación: {e}'
            }

# CLI y ejemplos
def run_predefined_examples(classifier: CloudModelClassifier):
    examples = [
        "Tiene todos los lenguajes y herramientas para poder programar un sistema.",
        "Proporciona instancias de servidores virtuales con almacenamiento y redes.",
        "Plataforma que facilita despliegue de aplicaciones con CI/CD y runtimes.",
        "Ejecuta funciones sin servidor activadas por eventos (lambda).",
        "Suite de productividad con suscripción para usuarios finales."
    ]
    for t in examples:
        res = classifier.classify_text(t)
        print("="*60)
```

```
        print("Texto:", t)
        print("Clasificación:", res['classification'], "Confianza:",
res['confidence'])
        print("Razonamiento:", res['reasoning'])
        print("Detalles:", res['details'])
    print("="*60)

def main():
    parser = argparse.ArgumentParser(description="Clasificador mejorado de modelos de
nube (IaaS/PaaS/FaaS/SaaS)")
    parser.add_argument("-t", "--text", type=str, help="Texto a clasificar (entre
comillas)")
    parser.add_argument("-e", "--examples", action="store_true", help="Ejecutar
ejemplos predefinidos")
    args = parser.parse_args()

    classifier = CloudModelClassifier()

    if args.examples:
        run_predefined_examples(classifier)
        return

    if args.text:
        res = classifier.classify_text(args.text)
        print("="*60)
        print("Texto:", args.text)
        print("Clasificación:", res['classification'])
        print("Confianza:", res['confidence'])
        print("Puntuaciones:", res['scores'])
        print("Razonamiento:", res['reasoning'])
        print("Detalles (raw):")
        for k, v in res.get('details', {}).items():
            print(f" - {k.upper()}: {v}")
        print("="*60)
        return

    # Modo interactivo simple
    print("Modo interactivo. Escribe 'salir' para terminar.")
    while True:
        try:
            text = input("\nEscribe descripción: ").strip()
            if not text:
                print("Texto vacío. Intenta otra vez.")
                continue
            if text.lower() in ('salir', 'exit', 'quit'):
                break
            res = classifier.classify_text(text)
            print("Clasificación:", res['classification'], "Confianza:",
res['confidence'])
            print("Razonamiento:", res['reasoning'])
        except KeyboardInterrupt:
            break

if __name__ == '__main__':
```

```
main()
```

A continuación se anexan screenshots de su funcionamiento:





Usar inteligencia artificial (IA) para crear la arquitectura de un clasificador de modelos de nube fue una experiencia que me sorprendió bastante. Incorporé conceptos avanzados que no conocía antes, como las coincidencias difusas para manejar texto que no siempre es exacto. Normalmente, aprender y aplicar estas técnicas toma mucho tiempo y requiere conocimiento específico, pero la IA me ayudó a integrarlas rápido y de forma ordenada.

Como estudiante de tecnologías computacionales, sé que muchas veces no podemos explotar todas las herramientas o técnicas por falta de tiempo, recursos o simplemente por no tener el conocimiento suficiente. La IA funciona como un apoyo que amplía nuestras capacidades y nos ayuda a superar esas limitaciones, permitiendo que el proyecto avance con mayor calidad y en menos tiempo.

En este proyecto, la IA no solo generó un código base eficiente, sino que también añadió un razonamiento claro que ayuda a entender por qué se clasificó un texto de cierta manera, lo cual es súper útil. Esto me dejó claro que la IA no reemplaza nuestro trabajo, sino que nos potencia, dejándonos concentrarnos en validar y mejorar lo que genera.

Conclusión

En resumen, usar IA en este tipo de proyectos me hizo ver lo mucho que podemos lograr cuando combinamos la tecnología con nuestra creatividad y conocimientos, incluso cuando el tiempo o recursos son limitados.