



Universidad de las Fuerzas Armadas ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería en Software

Desarrollo de Software Seguro

Laboratorio 1

Docente: Geovanny Cudco

25 de noviembre de 2025

1. Tema

Aplicación de Minería de Datos en el Desarrollo de Software Seguro

2. Contexto

La minería de datos (data mining) es una herramienta poderosa para mejorar la seguridad en el desarrollo de software, ya que permite analizar grandes volúmenes de datos relacionados con código fuente, historiales de vulnerabilidades, métricas de código y registros de incidentes. Esta actividad de investigación se centra en explorar cómo la minería de datos puede identificar patrones de riesgo, predecir vulnerabilidades y automatizar la detección en entornos de desarrollo ágiles. El objetivo es desarrollar un enfoque sistemático para integrar técnicas de minería de datos en procesos de DevSecOps (Development, Security, Operations), reduciendo el tiempo de exposición a amenazas y mejorando la calidad del software.

La actividad se estructurará siguiendo la metodología SEMMA (Sample, Explore, Modify, Model, Assess), que es un proceso iterativo y estructurado para proyectos de minería de datos, promovido por SAS Institute.

3. Objetivo

- Investigar fuentes de datos relevantes, como repositorios de código (e.g., GitHub), bases de datos de vulnerabilidades (e.g., CVE - Common Vulnerabilities and Exposures) y métricas de código estático (e.g., complejidad ciclomática, líneas de código, dependencias).

- Aplicar algoritmos de minería de datos para extraer patrones que indiquen potenciales vulnerabilidades (e.g., inyecciones SQL, fugas de memoria o configuraciones inseguras).
- Evaluar la efectividad de los modelos en escenarios reales de desarrollo software.
- Proponer una integración práctica en pipelines de CI/CD (Continuous Integration/Continuous Deployment).

4. Conjunto de Algoritmos de Minería de Datos aplicables

En el contexto de desarrollo de software seguro, se pueden aplicar algoritmos de minería de datos para tareas como clasificación (predecir si un módulo de código es vulnerable), clustering (agrupar códigos similares con riesgos comunes), detección de anomalías (identificar comportamientos inusuales) y reglas de asociación (encontrar correlaciones entre prácticas de codificación y vulnerabilidades). A continuación se presenta un conjunto relevante, seleccionado por su utilidad en análisis de código y predicción de riesgos:

- **Algoritmos de Clasificación (para predecir vulnerabilidades basadas en características del código):**
 - **Árboles de Decisión (Decision Trees):** Fáciles de interpretar, útiles para clasificar código como “vulnerable” o “seguro” basado en métricas como profundidad de anidamiento o uso de funciones inseguras.
 - **Bosques Aleatorios (Random Forest):** Mejora la precisión al combinar múltiples árboles, ideal para manejar datos desbalanceados (e.g., más código seguro que vulnerable).
 - **Máquinas de Vectores de Soporte (SVM - Support Vector Machines):** Efectivo para clasificar patrones complejos en código, como detección de inyecciones o *overflows*.
 - **Redes Neuronales (Neural Networks, incluyendo Deep Learning):** Para modelar relaciones no lineales en grandes *datasets* de código fuente, como en análisis de *AST* (Abstract Syntax Trees).
- **Algoritmos de Clustering (para agrupar patrones de riesgo):**
 - **K-Means:** Agrupa módulos de código similares por características (e.g., similitud en patrones de errores históricos), permitiendo identificar *clusters* de alto riesgo.
 - **Clustering Jerárquico (Hierarchical Clustering):** Útil para explorar jerarquías en vulnerabilidades, como en dependencias de bibliotecas.
- **Algoritmos de Reglas de Asociación (para descubrir correlaciones):**
 - **Apriori:** Encuentra reglas como “si se usa función X con parámetro Y, entonces hay alta probabilidad de vulnerabilidad Z”, basado en análisis de transacciones de commits en repositorios.

- **Algoritmos de Detección de Anomalías** (para identificar *outliers* en código):
 - **Isolation Forest:** Detecta anomalías en métricas de código que podrían indicar vulnerabilidades no vistas previamente.
 - **One-Class SVM:** Enfocado en aprender patrones “normales” de código seguro y alertar sobre desviaciones.

Estos algoritmos se eligen por su compatibilidad con datos de software (e.g., textuales, numéricos y categóricos) y su integración con bibliotecas como scikit-learn en Python. Se recomienda empezar con clasificación supervisada para predicción directa de vulnerabilidades.

4.1. Desarrollo de la actividad mediante SEMMA

La metodología SEMMA guiará el proceso de manera iterativa, asegurando que el análisis sea riguroso y reproducible. Cada fase se aplica al dataset de código y vulnerabilidades:

1. **Sample (Muestreo):** Selecciona una muestra representativa de los datos. Por ejemplo, extrae un subconjunto de repositorios *open-source* de GitHub (e.g., 10,000 commits) y bases de datos como *NVD* (National Vulnerability Database). Asegura diversidad en lenguajes (e.g., Java, Python, C++) y tipos de vulnerabilidades. Usa técnicas como muestreo estratificado para balancear clases (vulnerable vs. no vulnerable).
2. **Explore (Exploración):** Analiza los datos para identificar patrones iniciales. Calcula estadísticas descriptivas (e.g., frecuencia de vulnerabilidades por tipo de código), visualiza distribuciones (e.g., histogramas de métricas de complejidad) y detecta correlaciones (e.g., entre líneas de código y *bugs*). Herramientas como `pandas` y `matplotlib` en Python son ideales aquí.
3. **Modify (Modificación):** Limpia y transforma los datos. Elimina ruido (e.g., commits irrelevantes), maneja valores faltantes, normaliza métricas y extrae características (*feature engineering*), como vectores *TF-IDF* para texto de código o *embeddings* de código con modelos preentrenados (e.g., *CodeBERT*). Esto prepara los datos para el modelado.
4. **Model (Modelado):** Construye y entrena modelos usando los algoritmos listados. Por ejemplo, entrena un *Random Forest* para predecir vulnerabilidades basadas en características extraídas. Prueba combinaciones (e.g., *ensemble* de SVM y *Neural Networks*) para mejorar la precisión.
5. **Assess (Evaluación):** Evalúa los modelos con métricas como precisión, *recall*, *F1-score* y *ROC-AUC*, usando validación cruzada. Compara contra *baselines* (e.g., reglas estáticas de herramientas como *SonarQube*). Itera si es necesario, volviendo a fases anteriores para refinar.

El proceso SEMMA se repite hasta lograr un modelo robusto, con énfasis en la interpretabilidad para que los desarrolladores entiendan las alertas.

5. Entregable

Como entregable, se propone un modelo predictivo de vulnerabilidades basado en los algoritmos seleccionados (e.g., Random Forest para clasificación). Este modelo se integra en un pipeline de desarrollo como Jenkins o GitHub Actions mediante hooks o plugins (e.g., usando GitHub Actions workflows con scripts en Python).

5.1. Funcionamiento

- En cada *commit* o *pull request*, el *pipeline* extrae características del código fuente (e.g., mediante análisis de *AST parsing*).
- El modelo analiza patrones de riesgo (e.g., uso de funciones *deprecated* o patrones de inyección).
- Genera alertas automáticas (e.g., *issues* en GitHub o notificaciones en Slack) si se detecta una probabilidad de vulnerabilidad superior al 70 %.
- **Ejemplo de integración:** una acción en GitHub que ejecuta el modelo con `scikit-learn` y envía reportes HTML con explicaciones (usando SHAP para interpretabilidad).