MANUAL DE USUARIO PARA EL SISTEMA DE PREDICCIÓN DE ERRORES EN EL DESARROLLO DE SOFTWARE UTILIZANDO MACHINE LEARNING

Tabla de Contenidos

Introducción

El presente Manual de Usuario para el Sistema de Predicción de Errores en el Desarrollo de Software Utilizando Machine Learning ha sido diseñado con el propósito de proporcionar una guía clara y detallada para los usuarios que deseen implementar y aprovechar las capacidades de esta herramienta. Este sistema, basado en técnicas de aprendizaje automático, permite predecir errores en el desarrollo de software, optimizando la calidad del código y reduciendo los costos asociados a la corrección de defectos. Se busca orientar al usuario sobre los requisitos técnicos necesarios, el proceso de instalación, el uso del sistema para el entrenamiento y evaluación de los modelos, así como la interpretación de resultados obtenidos y la resolución de problemas comunes en el sistema.

Requisitos

Para ejecutar este proyecto, necesitarás tener instalado lo siguiente:

- **Sistema operativo**: Linux, macOS o Windows.
- **Python**: Versión 3.8 o superior, con el gestor de paquetes *pip* instalado para manejar dependencias.
- **Entorno virtual**: Se recomienda usar venv o virtualenv para aislar las dependencias del sistema y evitar conflictos entre paquetes.
- **Hardware**:
    - **RAM**: mínimo 8 Gb, recomendado 16 Gb.
    - **CPU**: mínimo 4 núcleos para un rendimiento aceptable, recomendado con 8 núcleos para procesar al datasets por completo.
- **Dataset**: Acceso al dataset Defectors (disponible en Zenodo) descomprimido. Las carpetas *line_bug_prediction_splits* y *jit_bug_prediction_splits* deben estar al mismo nivel que el archivo *main.py*, conteniendo los archivos *train.parquet*, *test.parquet* y *val.parquet*.

- **Dependencias:** Las bibliotecas necesarias para levantar el proyecto:
    - **Pandas** y **Numpy**: Para manipulación y análisis de datos.
    - **Scikit-learn**: Para modelos de machine learning y preprocesamiento.
    - **Matplotlib:** Para visualización de datos.
    - **Joblib**: Para facilitar la serialización de objetos de Python.
    - **XGBoost**: Para implementar algoritmos de aprendizaje automático.
    - **Imbalanced-learn**: Para trabajar con conjuntos de datos desbalanceados.
    - **Pyarrow**: Para optimizar memoria y evitar cuellos de botella en procesamiento.

Instalación

El proceso de instalación es sencillo y consta de los siguientes pasos detallados para garantizar una configuración correcta:

1. Clonar el repositorio

a. Descarga el código fuente del sistema desde el repositorio oficial de github. Usa el siguiente comando en una terminal:

```
> git clone https://github.com/fera1991/Defect-Predictor.git
> cd Defect-Predictor
```

b. Para descargar y clonar el repositorio es necesario tener instalado **git**.

2. Crear un entorno virtual

a. Crea un entorno virtual para aislar las dependencias del sistema. Esto evita conflictos con otras versiones de bibliotecas instaladas globalmente. Ejecutando el siguiente comando:

```
> python -m venv venv
```

```
> venv\Scripts\activate
```

b. Al activar el entorno, tu terminal mostrará (**venv**) o similar, indicando que estás dentro del entorno virtual.

3. Instalar dependencias

a. Instala las bibliotecas requeridas ejecutando:

```
> pip install numpy pandas matplotlib joblib scikit-learn xgboost imbalanced-learn pyarrow
```

b. Asegurarse de que **pip** esté actualizado

```
> pip install --upgrade pip
```

c. para evitar problemas durante la instalación.

4. Descargar y organizar los datos

a. Descarga el dataset Defectors desde Zenodo (Defectors: A Large Scale Python Dataset for Defect Prediction).

b. Descomprime el dataset en tu máquina.

c. Asegurarse de que las carpetas **line_bug_prediction_splits** y **jit_bug_prediction_splits** estén ubicadas en el mismo directorio que el archivo **main.py**:

```
/Defect-Predictor/
├── main.py
├── line_bug_prediction_splits/
│   ├── random/
│   │   ├── train.parquet
│   │   ├── test.parquet
│   │   ├── val.parquet
├── jit_bug_prediction_splits/
│   ├── random/
│   │   ├── train.parquet
│   │   ├── test.parquet
│   │   ├── val.parquet
```

d. Verifica que las subcarpetas */random/* de ambas carpetas contengan los archivos **train.parquet**, **test.parquet** y **val.parquet**.

5. Verificación inicial

a. Antes de ejecutar el sistema, asegúrate de que Python 3.8+ esté instalado (*python --version*).

b. Confirma que el entorno virtual está activado y que las dependencias se instalaron correctamente ejecutando:

```
> pip list
```

c. Revisa que las carpetas de datos estén en la ubicación correcta.

Uso del Sistema

Entrenamiento de Modelos

1. Ejecutar el script principal

```
> python main.py
```

Este comando inicia un flujo completo que incluye:

- **Carga de datos**: Lee los archivos Parquet desde las carpetas line_bug_prediction_splits/random/ y jit_bug_prediction_splits/random/, ubicadas al mismo nivel que main.py. El dataset **Defectors** debe estar descomprimido.
- **Procesamiento**: Fusiona datasets, asigna etiquetas binarias (0 para commits sin defectos, 1 para commits con defectos) y aplica balanceo si está configurado.
- **Extracción de características**: Genera las métricas de las ingenierías de características numérica, clasificatorias y textuales para el entrenamiento de los modelos.
- **Entrenamiento**: Entrena tres modelos (Random Forest, XGBoost y Voting Classifier) con búsqueda de hiperparámetros optimizada si está habilitada.
- **Evaluación inicial**: Calcula métricas de rendimiento en el conjunto de validación y genera visualizaciones (curvas ROC, matrices de confusión, histogramas de probabilidades).

2. Argumentos de línea de comandos

El script main.py permite personalizar la ejecución mediante los siguientes argumentos:

Argumento	Descripción	Tipo	Valor por Defecto	Restricciones	Ejemplo de Uso
--random_state	Semilla para reproducibilidad de resultados.	Entero	42	Entero positivo	--random_state 123
--data_type	Tipo de datos: 0 (sin balancear) o 1 (balanceado).	Entero	1	0 o 1	--data_type 0
--enable_hyperparam_search	Activa/desactiva la búsqueda de hiperparámetros (true/false).	Booleano	Definido en config.py	true o false	--enable_hyperparam_search false
--data_fraction	Fracción de datos a usar para entrenamiento (entre 0 y 1).	Flotante	1.0	Entre 0 y 1	--data_fraction 0.5

- Para ver todos los argumentos disponibles y sus valores por defecto, ejecuta:

```
> python main.py --help
```

- Ejemplo de ejecución personalizada:

```
> python main.py --random_state 2025 --data_type 1
--enable_hyperparam_search false --data_fraction 0.75
```

3. Resultados generados

Los resultados se guardan en ejecuciones/<timestamp>/, donde <timestamp> es la fecha y hora de ejecución.

- **Modelos entrenados**: Archivos .joblib (random_forest.joblib, xgboost.joblib, voting_classifier.joblib)
- **Gráficos**: Curvas ROC , histogramas de probabilidades, matrices de confusión, comparaciones de tiempos de entrenamiento y importancia de Características.
- **Métricas**: Reportes de clasificación, F1-scores , AUC-ROC y análisis por repositorio en archivos CSV.

Evaluación de Modelos Existentes

1. Ejecutar el script de evaluación:

Para evaluar modelos previamente entrenados, ejecuta el script evaluate_models.py, especificando el directorio donde se encuentran los modelos:

```
> python evaluate_models.py --models_dir ejecuciones/<timestamp>/
```

Este comando carga los modelos desde el directorio especificado (e.g., ejecuciones/<timestamp>/) y los evalúa en los conjuntos de prueba y validación, generando métricas y gráficos similares a los del entrenamiento. Los resultados se guardan en *evaluaciones/<timestamp>/*

2. Argumentos de línea de comandos:

El script evaluate_models.py permite personalizar la evaluación mediante los siguientes argumentos:

Argumento	Descripción	Tipo	Valor por Defecto	Restricciones	Ejemplo de Uso
--models_dir	Directorio donde se encuentran los modelos preentrenados (obligatorio).	Texto	Ninguno	Directorio válido con archivos .joblib	--models_dir ejecuciones/2025-07-19_18-08-00/
--random_state	Semilla para reproducibilidad de resultados.	Entero	42	Entero positivo	--random_state 123
--data_type	Tipo de datos: 0 (sin balancear) o 1 (balanceado).	Entero	1	0 o 1	--data_type 0
--fixed_threshold	Umbral fijo para clasificación (opcional, entre 0 y 1).	Flotante	None	Entre 0 y 1, o None	--fixed_threshold 0.5
--val_percent	Porcentaje del conjunto de validación a usar (entre 0 y 1).	Flotante	0.2	Entre 0 y 1	--val_percent 0.3

--test_percent	Porcentaje del conjunto de prueba a usar (entre 0 y 1).	Flotante	0.2	Entre 0 y 1	--test_percent 0.3

Para ver todos los argumentos disponibles y sus valores por defecto, ejecuta:

```
> python evaluate_models.py --help
```

Ejemplo de ejecución personalizada:

```
> python evaluate_models.py --models_dir
ejecuciones/2025-07-19_18-08-00/
 --random_state 2025 --data_type 1 --fixed_threshold 0.5
--val_percent 0.3 --test_percent 0.3
```

3. Resultados generados:

Los resultados se guardan en *evaluaciones/<timestamp>/*, donde *<timestamp>* es la fecha y hora de ejecución.

- **Métricas**: Reportes de clasificación, *F1-scores*, comparación detallada de modelos, y métricas con umbral.
- **Gráficos**: Curvas *ROC* para los conjuntos de prueba y validación.

Interpretación de Resultados

El sistema genera una variedad de resultados que ayuda a entender el rendimiento de los modelos. A continuación, se explican en detalle los principales resultados y cómo interpretarlos:

- **Reportes de Clasificación**:
  - Incluyen precisión, recall, F1-score y soporte para las clases "Sin Defectos" (0) y "Con Defectos" (1).
  - El F1-score es particularmente útil para evaluar el equilibrio entre precisión y recall en la clase positiva.
- **Curvas ROC**:
  - Muestran el área bajo la curva (AUC-ROC). Un valor cercano a 1 indica un modelo con alta capacidad de discriminación.
- **Matriz de Confusión**:
  - Detalla los verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos, ayudando a entender los errores de predicción.
- **F1-scores por Repositorio**:
  - Analiza el rendimiento del modelo en diferentes repositorios, útil para identificar proyectos con mayor incidencia de defectos.
- **Importancia de Características**:
  - Indica qué características tienen mayor impacto en las predicciones (e.g., loc_full, token_entropy_full, num_error_keywords_diff).

Solución de Problemas

El sistema incluye mecanismos para diagnosticar y resolver problemas comunes. A continuación, se detallan los errores más frecuentes, sus causas y soluciones:

- **Error de memoria**:
    - **Causa:** El procesamiento de datos o la búsqueda de hiperparametros que consume demasiada memoria RAM, más cuando son dataset grandes
    - **Solución:**
        - Reducir la fracción de datos en el argumento *--data_fraction* o modificando directamento en el archivo **config.py** para procesar menos datos.
        - Disminuye el número de iteraciones en la búsqueda de hiperparametros en **train.py**
- **Modelos no encontrados**:
    - **Causa**: El script **evaluate_models.py** no encuentra los archivos **.joblib** en la dirección ingresada.
    - **Solución:**
        - Verificar que la carpeta contenga los modelos entrenados ejemplo: *ejecuciones/2025-07-15/randomforest_pipeline_model.joblib*
        - Revisar los logs de **traing.log** para confirmar que el entrenamiento se completó sin errores.
- **Resultados inconsistentes**:
    - **Causa:** Datos corruptos, configuraciones incorrectas o desbalanceo excesivo en el dataset.
    - **Solución:**
        - Verificar configuración en **config.py**, especialmente **DATA_TYPE** y **TEST_CLASS1_RATIO**
        - Revisa los logs en **train.log** y **evaluate.log** para identificar errores específicos.
- **Tiempo de ejecución prolongado**:
    - **Causa**: La búsqueda de hiperparamétricos o el procesamiento de texto (TF-IDF, SVD) puede ser intensivo.
    - **Solución:**
        - Reducir el valor de **n_iter** en **train.py.**

- Disminuye el valor **max_features** en **TFDIF_PARAMS** o **SVD_PARAMS** en **config.py:**

```
> TFIDF_PARAMS = {
      'max_features': 5000,  # Reduce de 12000 a 5000
      'ngram_range': (1, 2)
}
SVD_PARAMS = {
      'n_components': 20  # Reduce de 50 a 20
}
```

- Usar un procesador más potente o ejecutar el sistema en un entorno con múltiples núcleos, aumentando **n_jobs** en config.py.