MALWARE CLASSIFICATION USING STATIC DISASSEMBLY & MACHINE LEARNING

STUDENT NAME:

Wahaj Javed Alam (20K-0208)

RESEARCH PAPER BY:

Zhenshuo Chen

Tomas Ward

Eoin Brophy

Understanding Malware

Malware refers to any malicious software program that is intentionally designed to harm, compromise, or exploit computer systems or data without the user's consent.

Some common malware examples include viruses, worms, trojans, rootkits, spywares.

Malware can be propagated through email attachments, infected files downloaded over tor networks, via phishing links

Malware Analysis

There are two types of malware analysis/detection techniques.

1. Static Analysis (Signature Based analysis)
2. Dynamic Analysis
    a. Sandboxing
    b. Deep learning/ Generative AI Methods

Static Based Malware Analysis

A signature refers to a unique identifier, created by security researchers and antivirus companies, which is associated with a particular malware variant.

Malware authors use randomization, encryption and obfuscation techniques to modify malware signature and evade detection.

The drawback is that new unseen malware is not detected.

Abstract

Develop a simple Malware Classifier which is resource efficient and does the job over presented data extremely well

TERMINOLOGIES

REVERSE ENGINEERING

Malware reverse engineering is the process of analyzing malicious software to understand its functionality, origin, and purpose.

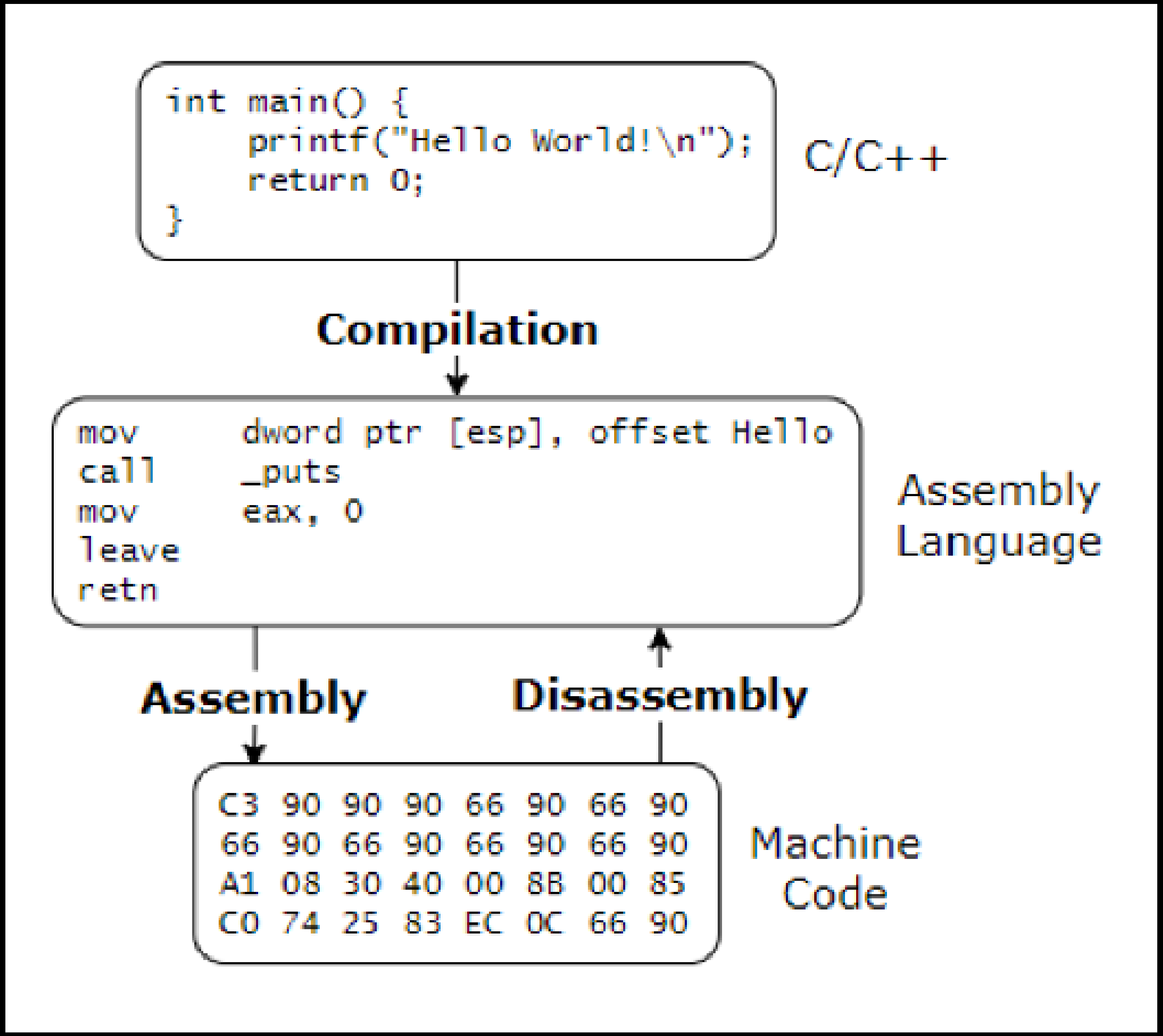
ASSEMBLY

The closest language to machine code understandable by humans is Assembly Language which goes through a process known Assembly to become machine code.

DISASSEMBLY

Disassembly which is the process of converting Executable files into assembly language

Compilation, Assembly and Disassembly Visualized



CODE OBFUSCATION

Code Obfuscation adds needless roundabouts are added to the code to obstruct and hinder the process of disassembly

NAME MANGLING

When code is compiled, the generated function names depend on compiler used, thus causing noise in API N-grams process

ENCRYPTION

Code Encryption packs and encrypts executable files on disk

LAZY LOADING

In order to counter the malwares being detected based on library imports, malware developers import sensitive libraries right before they need to use them, thus those imports don't go into the PE Headers

DATASET

Data Composition:

It is taken from **Microsoft Malware Classification Dataset on Kaggle**.

Raw Data

The Dataset Contains ASM Files and Bytes Files for each malware

Metadata Manifest

Insights into the binary files' structure and behavior.

DATASET DESCRIPTION

Class	Name	Type	Frequency
1	Ramnit	Worm	1541
2	Lollipop	Adware	2478
3	Kelihos_ver3	Backdoor	2942
4	Vundo	Trojan	475
5	Simda	Backdoor	42
6	Tracur	Trojan Downloader	751
7	Kelihos_ver1	Backdoor	398
8	Obfuscator.ACY	obfuscate malware	1228
9	Gatak	Backdoor	1013

PROPOSED FEATURES

FILE SIZE

Bytes file Size | ASM file Size | Ratio of ASM/BYTE

```
def extractFileSizes(fileIDs):
    df = pd.DataFrame(columns=["ID", "Asm-Size", "Byte-Size", "Ratio"], dtype=float).set_index("ID")
    for id in tqdm(fileIDs):
        df.at[id, "Asm-Size"] = os.path.getsize(TRAIN_DIRECTORY_PATH + id + ".asm")
        df.at[id, "Byte-Size"] = os.path.getsize(TRAIN_DIRECTORY_PATH + id + ".bytes")
    df[["Asm-Size", "Byte-Size"]] = df[["Asm-Size", "Byte-Size"]].astype(int)
    df["Ratio"] = (df["Asm-Size"] / df["Byte-Size"]).round(5)
    return df
```

API 4-GRAMS

Function and API Calls in ASM files

FUNCTION CALLS REGEX

```
functionCallsRegex = re.compile(r"\scall\s+(?:ds:)(?:__imp_)?([^\s]+)",
```

The Calling format looks like this:

call _memmove_s | call ds:VirtualAllocEx | call sub_6D1757D6

** The prefix ds should be removed from function names

API 4-GRAMS

Function and API Calls in ASM files

API CALLS REGEX

```
apiRegex = re.compile(r"extrn\s+(?:__imp_)?([^\s:]+)"
```

The Calling format looks like this:

extrn LoadResource:dword | extrn __imp_RtlUnwind:dword

** The prefix `__imp_` and data type `dword` should also be removed.

OPCODE 4-GRAMS

Disassembly Instructions in ASM Code

OPCODE REGEX

```
opcodeRegex = re.compile(r"\s[\dA-F]{2}(?:\+)?\s+([a-z]+)\s")
```

The OPCODE format looks like this:

6A 00	push 0	8B 4C 24 04	mov ecx, [esp+4]

IMPORT LIBRARY

Libraries imported from the IMPORT TABLE

LIBRARY REGEX

```
libraryRegex = re.compile(r"Imports\s+from\s+(.+).dll"
```

The library call format looks like this:

Imports from KERNEL32.dll | Imports from java.dll

PORTABLE EXECUTABLE

Extraction of PE Section Sizes and Permissions

Each Attribute in a section contains the following:

Size	Description
Virtual Size	The total size of the section when it is loaded into memory.
Raw Size	The size of the section or the size of the initialized data in the disk file.
Flags	Executable, readable and writable.

PORTABLE EXECUTABLE

Extraction of PE Section Sizes and Permissions

ATTRIBUTE FORMAT

```
.text:00401000    ; Section 1. (virtual address 00001000)
.text:00401000    ; Virtual size        : 0002964D ( 169549.)
.text:00401000    ; Section size in file : 00029800 ( 169984.)
.text:00401000    ; Offset to raw data for section: 00000400
.text:00401000    ; Flags 60000020: Text Executable Readable
```

CONTENT COMPLEXITY

Asm-Length	Zip-Asm-Len	Asm-Zip-Ratio
Byte-Length	Zip-Byte-Len	Byte-Zip-Ratio

```
    df.at[id, "Asm-Len"] = len(asm)
    df.at[id, "Zip-Asm-Len"] = len(zlib.compress(asm))
    df.at[id, "Byte-Len"] = len(bytes)
    df.at[id, "Zip-Byte-Len"] = len(zlib.compress(bytes))
df[["Asm-Len", "Zip-Asm-Len", "Byte-Len", "Zip-Byte-Len"]] =
                    df[["Asm-Len", "Zip-Asm-Len", "Byte-Len", "Zip-Byte-Len"]].astype(int)
df["Asm-Zip-Ratio"] = (df["Asm-Len"] / df["Zip-Asm-Len"]).round(5)
df["Byte-Zip-Ratio"] = (df["Byte-Len"] / df["Zip-Byte-Len"]).round(5)
return df
```

PROPOSED SOLUTION

COMPONENTS

DIMENSIONALITY REDUCTION

TPOT CLASSIFIER

SEABORN

MATPLOTLIB

TQDM

REGULAR EXPRESSIONS

SKLEARN

PANDAS

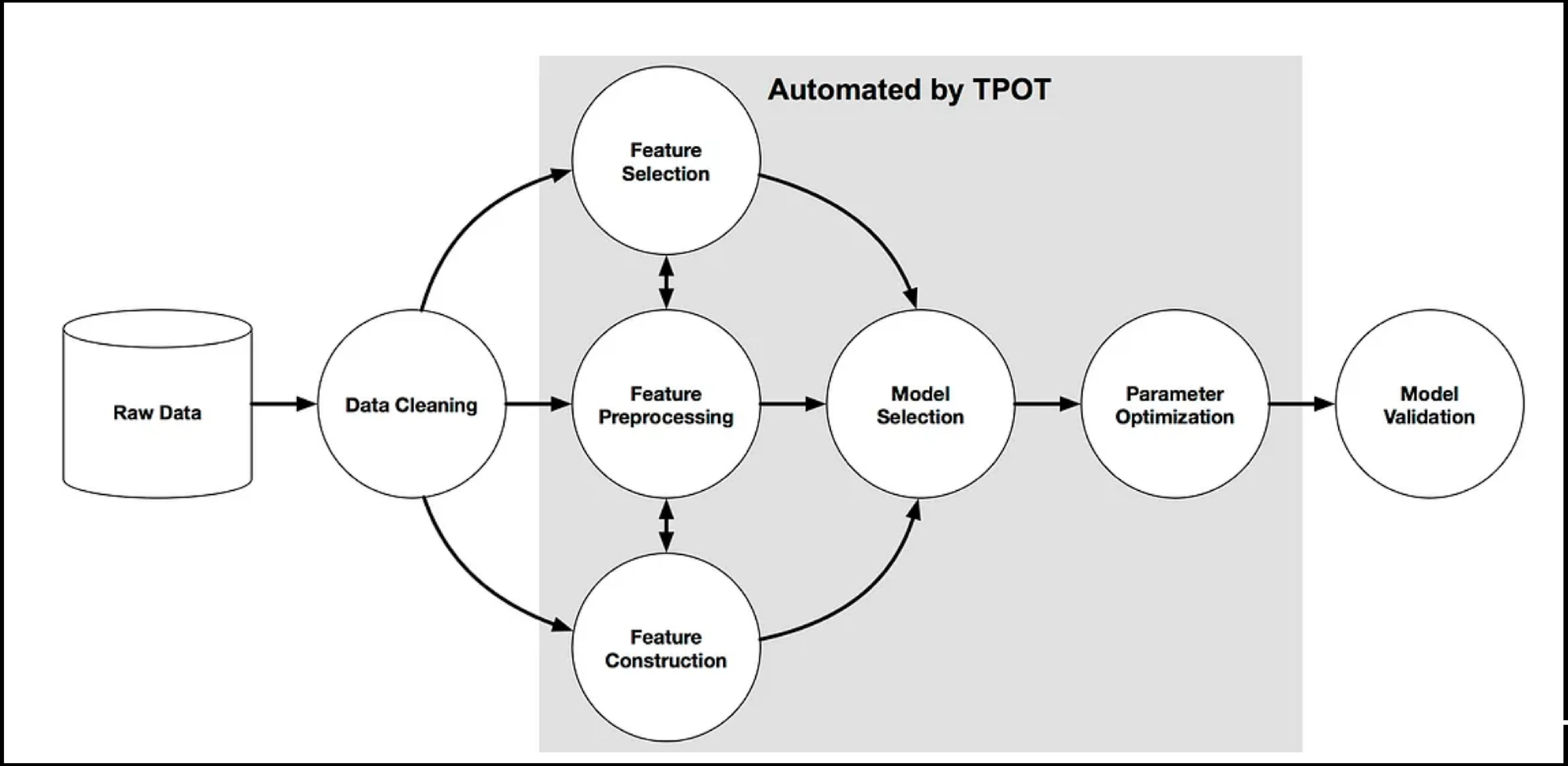
DIMENSIONALITY REDUCTION

Done by selecting the Top-N Features based on Frequency

SECTION SIZES	---------	846 -> 25
API 4-GRAM	---------	402972 -> 5000
OPCODE 4-GRAM	---------	1408515 -> 5000
IMPORT LIBRARY	---------	570 -> 300

TPOT CLASSIFIER

Tree Based Pipeline Optimization Tool



TPOT CLASSIFIER

Tree Based Pipeline Optimization Tool

For Classification purposes it scans through

- Linear models (like Logistic Regression)
- Naive Bayes models (like Bernoulli NB, Gaussian NB, MultinomialNB)
- Tree models (like Decision Tree Classifier)
- Ensemble models (like Random Forest Classifier)
- SVM models (like LinearSVC)
- XGBoost models

TPOT CLASSIFIER

Tree Based Pipeline Optimization Tool

TPOT uses genetic programming to explore a large search space of possible pipelines, including data preprocessing steps, feature selection, and the configuration of various machine learning models.

PARAMETERS SET:

- GENERATIONS = 5
- CROSS VALIDATION = 5-FOLD
- SCORING METRIC = ACCURACY

TPOT CLASSIFIER

Tree Based Pipeline Optimization Tool

TPOT uses genetic programming to explore a large search space of possible pipelines, including data preprocessing steps, feature selection, and the configuration of various machine learning models.

PARAMETERS SET:

- GENERATIONS = 5
- CROSS VALIDATION = 5-FOLD
- SCORING METRIC = ACCURACY

RESULTS

Feature	Classification Accuracy	Dimensions	Best Model
File Size	94.30%	3	Extra Trees Classifier
Section Size	98.07%	846 -> 25	Extra Trees Classifier
Section Permission	97.79%	9	Extra Trees Classifier
API 4-GRAM	57.27%	402972 -> 5000	Linear SVC Classifier
Content Compleixty	97.24%	6	Random Forest Classifier
Import Library	92.59%	570 -> 300	MLP Classifier
All Features	98.94%	43	Extra Trees Classifier

REC

THANK YOU