Team 2: Introduction to Data Mining
FA23_DATS6103 : Summary Report

Microsoft Malware Prediction

Introduction

In today's technological landscape, the omnipresence of malware, a shortened term for malicious software, has escalated into a critical threat, targeting data integrity, device security, and user privacy. This ominous software, engineered with nefarious intent, poses a persistent challenge to both individual users and enterprises. The exponential growth of the malware industry has led to the creation of sophisticated technologies, adept at circumventing traditional security measures, compelling the continual evolution of anti-malware solutions to counter these evolving threats. Microsoft, a stalwart in anti-malware development, actively deploys its robust utilities across a staggering network of over 150 million computers worldwide, generating a deluge of data exceeding tens of millions of daily points for potential malware analysis. The efficacy of combating this proliferation hinges on the precise analysis and classification of this colossal dataset, containing nine distinct classes of malware, ranging from Ramnit to Gatak, each necessitating precise identification. The dataset itself, comprising both .asm and .bytes files, presents a monumental challenge with its 200GB size, encompassing 10,868 files of each type, totaling 21,736 files. Translating the real-world challenge of malware detection into a machine learning endeavor entails accurately classifying these diverse malware types into their respective categories, constituting a multi-class classification problem. Performance evaluation relies on metrics like multi-class log-loss and the confusion matrix, reflecting the accuracy in identifying these varied malware types, while considering constraints such as latency and the imperative need for class probabilities. This convergence of cybersecurity challenges with machine learning methodologies symbolizes an ongoing pursuit to fortify digital defenses, mitigate vulnerabilities, and confront the ever-evolving landscape of malicious software.

Key-Words : Malware, malicious software, data security, device integrity, anti-malware solutions, Microsoft, cybersecurity, dataset, classification, multi-class, machine learning, log-loss, confusion matrix, threat, technology, analysis, detection, challenges, evolution, vulnerabilities, digital defenses

Smart Questions

- What specific features within the dataset exhibit strong predictive potential for identifying instances of malware infections?
- In what ways does the outcome of this project significantly bolster and elevate ongoing cybersecurity endeavors?
- Do the features encapsulated within the .asm and .bytes files provide comprehensive coverage for effective malware detection, or might additional data sources be necessary?
- Given the dataset's inherent imbalance, can strategic hyper-parameter tuning effectively address this issue, and which machine learning model demonstrates superior resilience and performance under these conditions?

Literature Review

The landscape of malware detection and classification has witnessed significant strides through notable contributions in academia and industry. Ahmadi et al.'s work, "NO to overfitting!", focused on feature extraction and fusion for effective malware family classification, stands as a pivotal piece addressing the persistent challenge of overfitting in machine learning-based detection systems. Collaborative research involving scholars from the University of Cagliari, Italy, Skolkovo Institute of Science and Technology, and Russian institutions underlines the global effort in this domain. Concurrently, Nataraj's exploration into visualizing and automatically classifying malware images from the Vision Research Lab at the University of California, Santa Barbara, provides innovative insights that can enhance malware understanding and detection mechanisms.

The Microsoft Malware Classification Challenge, accompanied by contributions from Trend Micro and individual researchers like ch0upi, miaoski, and Kyle Chung, has set a benchmark in the field. Their collective efforts have significantly advanced the understanding of malware behavior and detection methodologies, profoundly impacting cybersecurity practices. Moreover, resources such as the malware detection repository on GitHub curated by dchad offer valuable tools, datasets, and insights for researchers and practitioners in this arena.

These seminal works underscore the interdisciplinary nature of malware detection research, amalgamating machine learning, data visualization, and collaborative international efforts. Their advancements show promise in fortifying cybersecurity measures by addressing overfitting challenges, developing robust methodologies for identifying and classifying malware, ultimately bolstering defenses against the evolving landscape of cyber threats.

Description Of Data

The malware detection dataset presents a comprehensive collection featuring .asm and .bytes file formats per malware instance. .asm files hold assembly language code while .bytes files contain raw hexadecimal data, omitting the PE header. This sizable dataset comprises 200GB, with .bytes files occupying 50GB and .asm files totaling 150GB. With 10,868 instances of each file type, a total of 21,736 files span the dataset, showcasing nine diverse malware types: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, and Gatak. This dataset provides an extensive resource for crafting and evaluating robust malware detection models. Its breadth allows for in-depth exploration and pattern recognition across various malware classes, facilitating the development of sophisticated algorithms. These advancements aid in the accurate identification and classification of malicious entities, reinforcing cybersecurity protocols in response to evolving threats.

.asm File Example

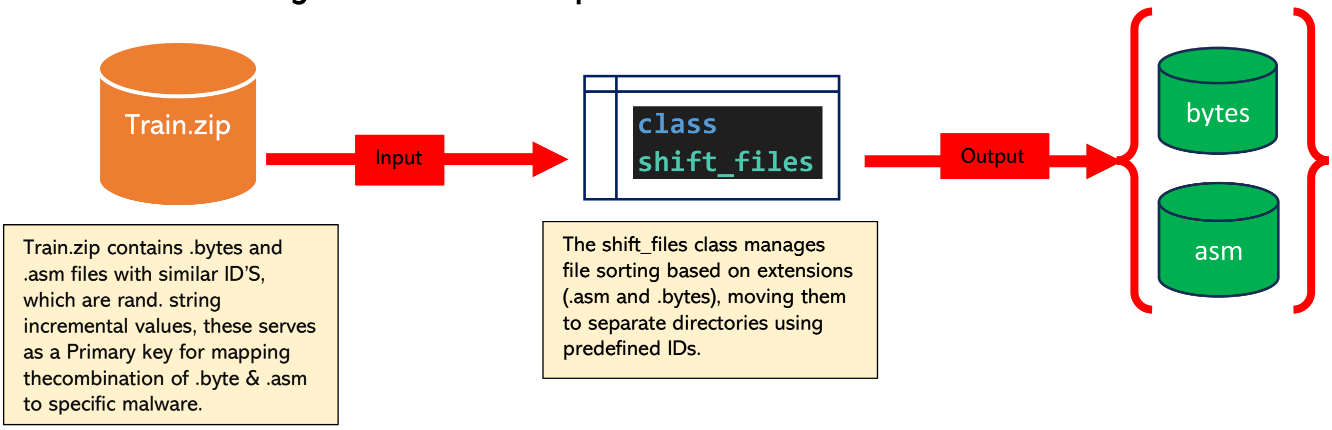
```
.text:00401000                                  assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:00401000 56                               push    esi
.text:00401001 8D 44 24 08                          lea     eax, [esp+8]
.text:00401005 50                               push    eax
.text:00401006 8B F1                                mov     esi, ecx
.text:00401008 E8 1C 1B 00 00                           call    ??0exception@std@@QAE@ABQBD@Z ; std::exception::exception(char const *
const &)
.text:0040100D C7 06 08 BB 42 00                        mov     dword ptr [esi], offset off_42BB08
.text:00401013 8B C6                                mov     eax, esi
.text:00401015 5E                               pop     esi
.text:00401016 C2 04 00                             retn    4
.text:00401016                          ; ---------------------------------------------------------------------------
.text:00401019 CC CC CC CC CC CC CC                     align 10h
.text:00401020 C7 01 08 BB 42 00                        mov     dword ptr [ecx], offset off_42BB08
.text:00401026 E9 26 1C 00 00                           jmp     sub_402C51
.text:00401026                          ; ---------------------------------------------------------------------------
.text:0040102B CC CC CC CC CC                           align 10h
.text:00401030 56                               push    esi
.text:00401031 8B F1                                mov     esi, ecx
.text:00401033 C7 06 08 BB 42 00                        mov     dword ptr [esi], offset off_42BB08
.text:00401039 E8 13 1C 00 00                           call    sub_402C51
.text:0040103E F6 44 24 08 01                           test    byte ptr [esp+8], 1
.text:00401043 74 09                                jz      short loc_40104E
.text:00401045 56                               push    esi
.text:00401046 E8 6C 1E 00 00                           call    ??3@YAXPAX@Z    ; operator delete(void *)
.text:0040104B 83 C4 04                             add     esp, 4
.text:0040104E
.text:0040104E                          loc_40104E:                     ; CODE XREF: .text:00401043j
.text:0040104E 8B C6                                mov     eax, esi
.text:00401050 5E                               pop     esi
.text:00401051 C2 04 00                             retn    4
.text:00401051                          ; ---------------------------------------------------------------------------
```

.byte File Example

```
00401000 00 00 80 40 40 28 00 1C 02 42 00 C4 00 20 04 20
00401010 00 00 20 09 2A 02 00 00 00 00 8E 10 41 0A 21 01
00401020 40 00 02 01 00 90 21 00 32 40 00 1C 01 40 C8 18
00401030 40 82 02 63 20 00 00 09 10 01 02 21 00 82 00 04
00401040 82 20 08 83 00 08 00 00 00 00 02 00 60 80 10 80
00401050 18 00 00 20 A9 00 00 00 00 04 04 78 01 02 70 90
00401060 00 02 00 08 20 12 00 00 00 40 10 00 80 00 40 19
00401070 00 00 00 00 11 20 80 04 80 10 00 20 00 00 25 00
00401080 00 00 01 00 00 04 00 10 02 C1 80 80 00 20 20 00
00401090 08 A0 01 01 44 28 00 00 08 10 20 00 02 08 00 00
004010A0 00 40 00 00 00 34 40 40 00 04 00 08 80 08 00 08
004010B0 10 00 40 00 68 02 40 04 E1 00 28 14 00 08 20 0A
004010C0 06 01 02 00 40 00 00 00 00 00 00 20 00 02 00 04
004010D0 80 18 90 00 00 10 A0 00 45 09 00 10 04 40 44 82
004010E0 90 00 26 10 00 00 04 00 82 00 00 00 20 40 00 00
004010F0 B4 00 00 40 00 02 20 25 08 00 00 00 00 00 00 00
00401100 08 00 00 50 00 08 40 50 00 02 06 22 08 85 30 00
00401110 00 80 00 80 60 00 09 00 04 20 00 00 00 00 00 00
00401120 00 82 40 02 00 11 46 01 4A 01 8C 01 E6 00 86 10
00401130 4C 01 22 00 64 00 AE 01 EA 01 2A 11 E8 10 26 11
00401140 4E 11 8E 11 C2 00 6C 00 0C 11 60 01 CA 00 62 10
00401150 6C 01 A0 11 CE 10 2C 11 4E 10 8C 00 CE 01 AE 01
00401160 6C 10 6C 11 A2 01 AE 00 46 11 EE 10 22 00 A8 00
00401170 EC 01 08 11 A2 01 AE 10 6C 00 6E 00 AC 11 8C 00
00401180 EC 01 2A 10 2A 01 AE 00 40 00 C8 10 48 01 4E 11
00401190 0E 00 EC 11 24 10 4A 10 04 01 C8 11 E6 01 C2 00
```

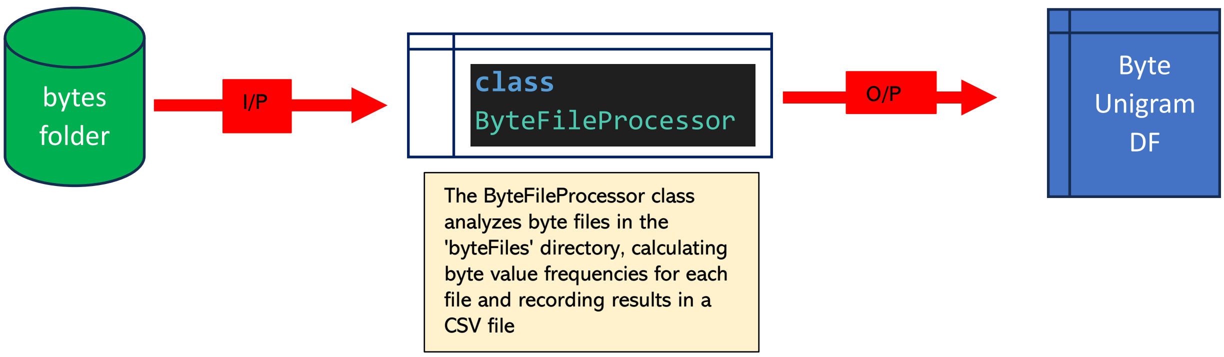
Data - Preprocessing

❖ Creating Folders for the Required Files



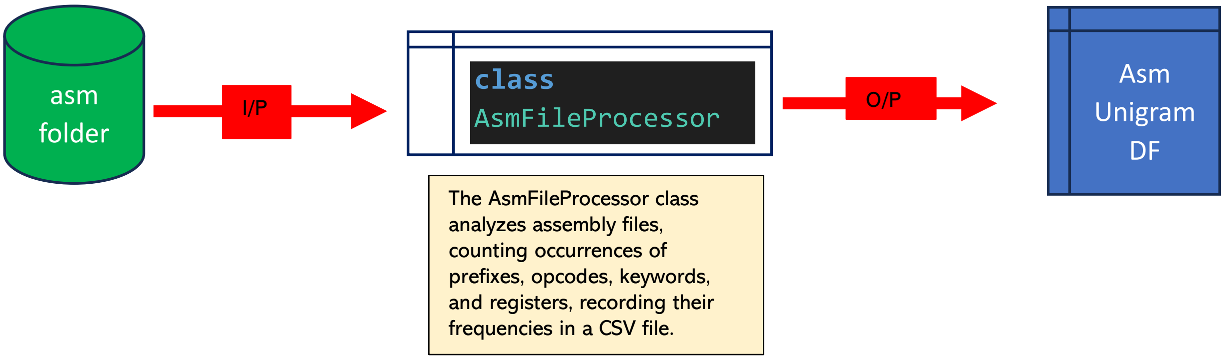
The 'shift_files' process partitions files from a source directory into two destinations based on their extensions (.asm and .bytes) and predefined IDs. It identifies and categorizes files by extension, separating them into two groups. Then, using predetermined IDs from a CSV file, it selects specific files to relocate to distinct destinations. This process aims to organize and segregate files for subsequent analysis or processing based on their file types and predetermined identifiers.

❖ Byte Unigram Vectorization



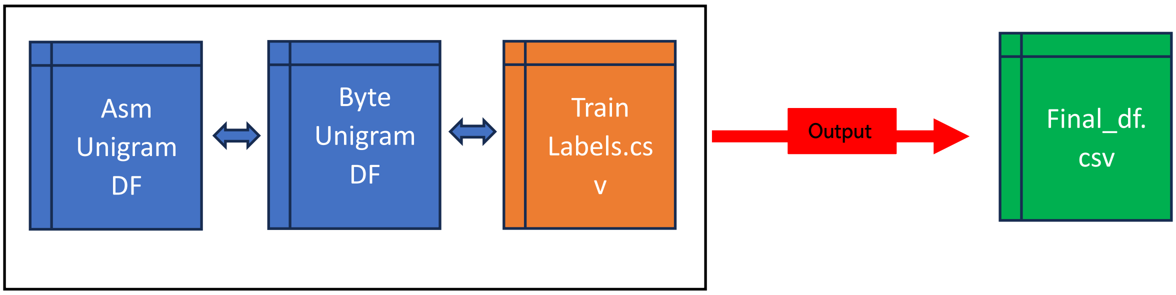
The ByteFileProcessor class contains a static method, process_byte_files(), designed to analyze byte files stored within the 'byteFiles' directory. This method reads each file, computes the frequency of every byte value present, and compiles these results into a CSV file. The process involves listing files in the 'byteFiles' directory, initializing a feature matrix to store byte frequency information, and writing headers to the output CSV file. For each file in the directory, the method reads and processes text files, counting occurrences of each byte value. Finally, it constructs a feature matrix and writes this matrix to the CSV file, facilitating a comprehensive analysis of byte frequency distributions within the stored files.

❖ Asm Unigram Vectorization



The AsmFileProcessor class offers static methods to scrutinize assembly (asm) files, delving into specific patterns like prefixes, opcodes, keywords, and registers. It quantifies their occurrences and records their frequencies in a CSV file. The firstprocess() method initializes lists of assembly language components for counting, reads and analyzes assembly files, and writes the counts of identified features to an output CSV file. It meticulously processes each file within the 'asmfiles' directory, calculating counts for prefixes, opcodes, keywords, and registers. The main() method orchestrates the execution of the file processing, calling the firstprocess() method within the AsmFileProcessor class. This systematic analysis provides a comprehensive breakdown of specific assembly language components within the files, offering a nuanced insight into their usage and frequency distributions.

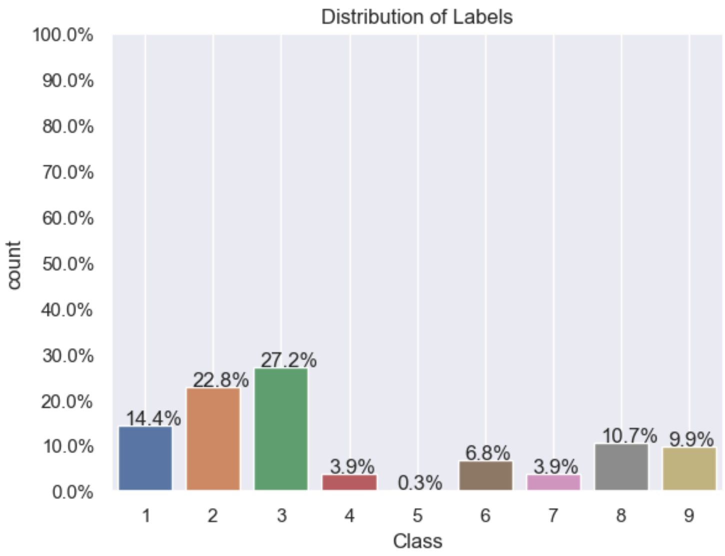
❖ Final Data Frame



It involves extracting column names from two DataFrames (byte_file and asm_file), merging these DataFrames based on a shared 'ID' column to create an integrated DataFrame (final_df), and further consolidating this result with another DataFrame (Train_labels) using the same 'ID' column for comprehensive data aggregation and analysis.

Exploratory Data Analysis

Class Distribution



Description

Each class (labeled 1 through 9) is associated with a specific proportion, reflecting its relative prevalence in the dataset. For instance, Class 3 constitutes the largest portion at approximately 27.16%, followed by Class 2 at about 22.83%, and Class 1 at 14.43%. Classes 8, 9, 6, 4, 7, and 5 exhibit decreasing proportions, highlighting their declining presence within the dataset, with Class 5 having the smallest representation at around 0.33%. This distribution offers insights into the dataset's class imbalance, crucial for understanding the varying degrees of representation among different classes, which can significantly impact machine learning model performance and require appropriate handling during training to ensure balanced and accurate predictions.

Relative Frequency Comparison of Dependent Variable with specific Independed Variable.

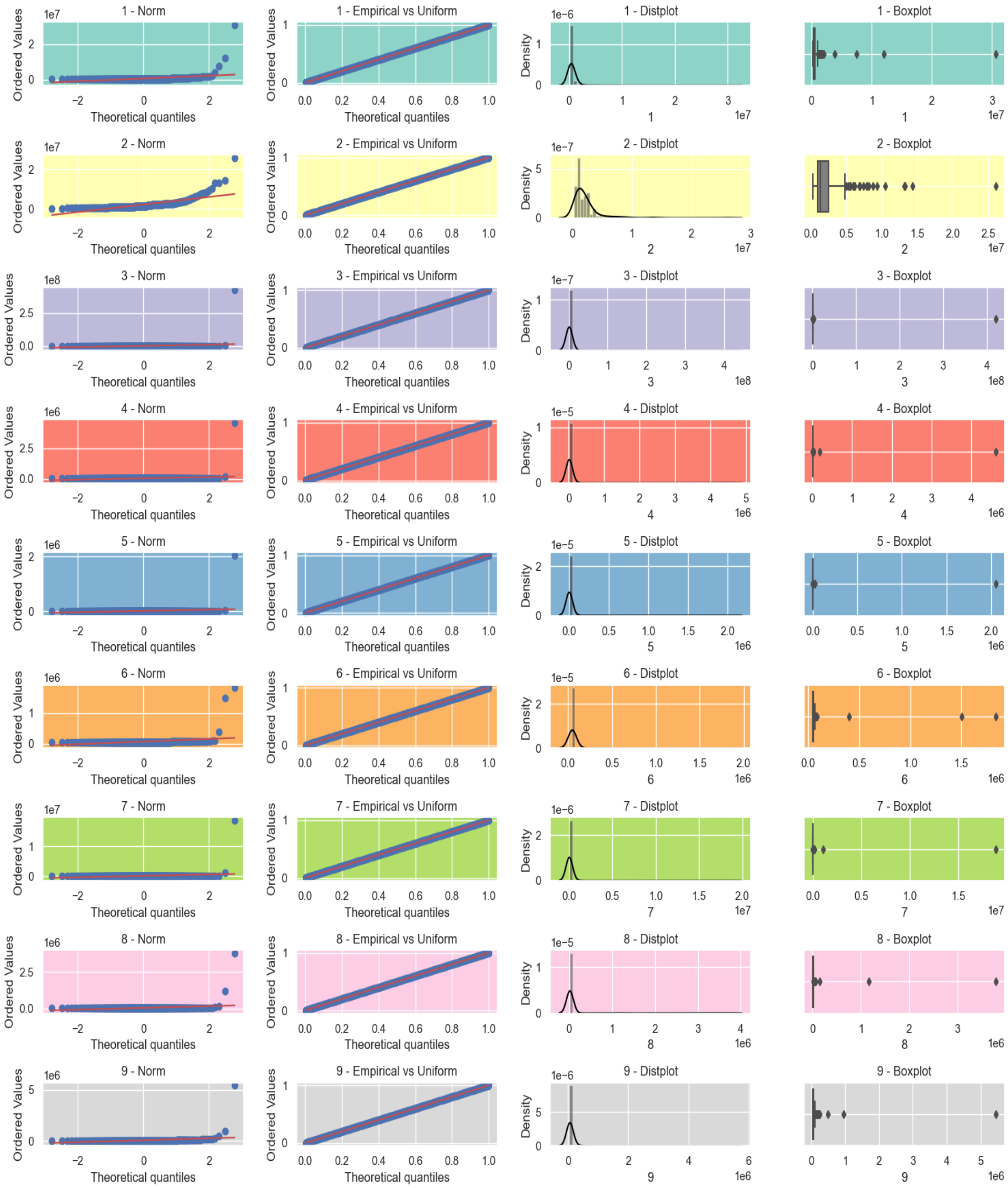
$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Comparison of these probabilities with 'Normal and Uniform'

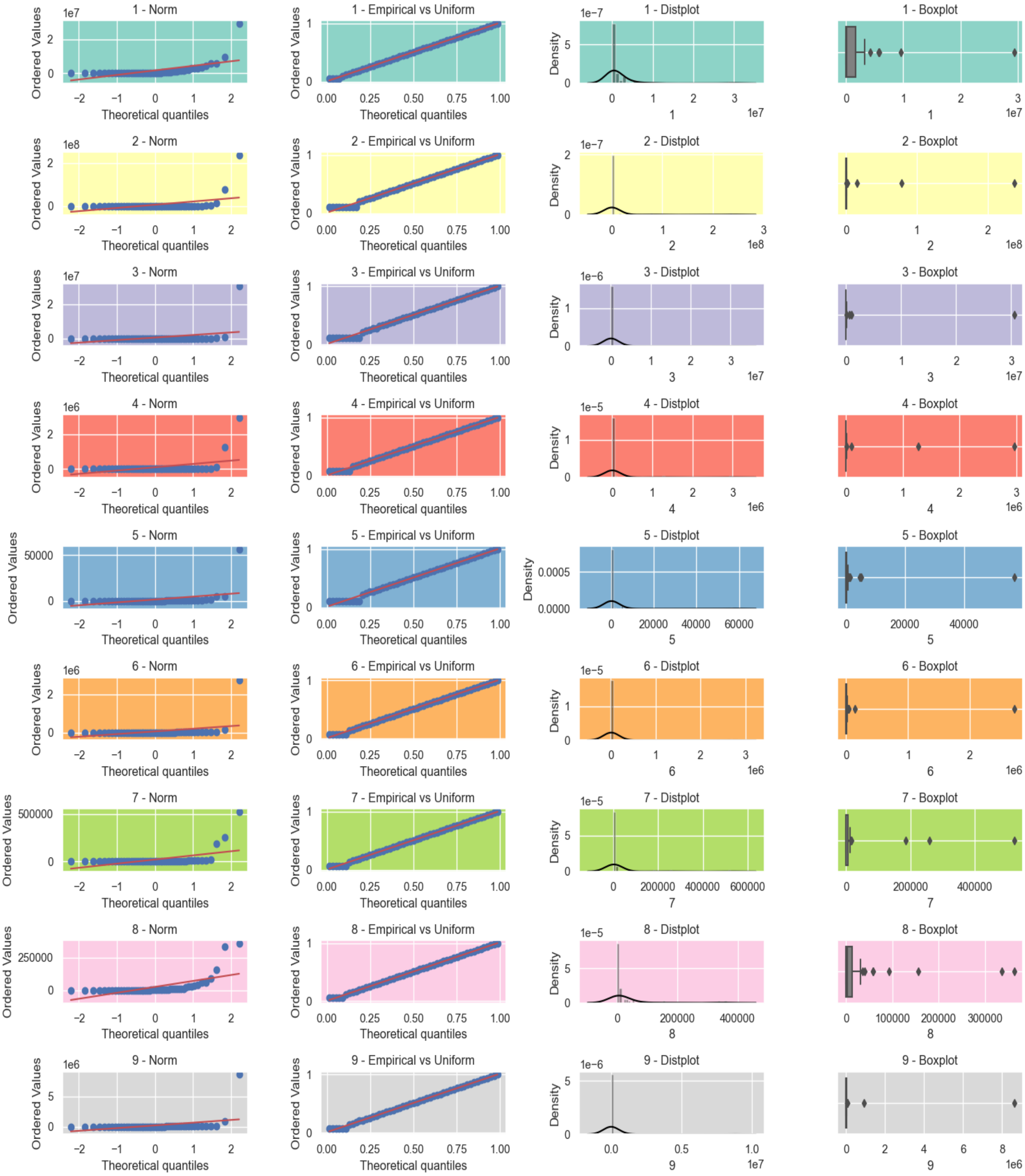
This process, named get_probs, calculates probabilities based on class counts derived from the final_df DataFrame. Initially, it checks and removes the 'ID' column from the lists asm_cols and byte_cols. Next, it groups the final_df DataFrame by the 'Class' column and applies a function (Analysis.get_counts) to obtain counts for specific columns.

After this aggregation, the code proceeds to calculate probabilities. It divides the counts of byte and assembly columns by the respective class values along the rows, transposes the resulting DataFrames (bytes_probs and asm_probs) for ease of interpretation, and returns these probability matrices. The aim is to obtain probabilities for each class across different byte and assembly features, aiding in understanding the distribution and likelihood of specific features occurring within each class in the dataset

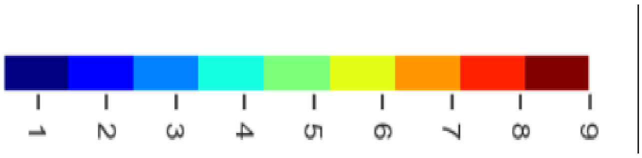
Bytes Files Comparison

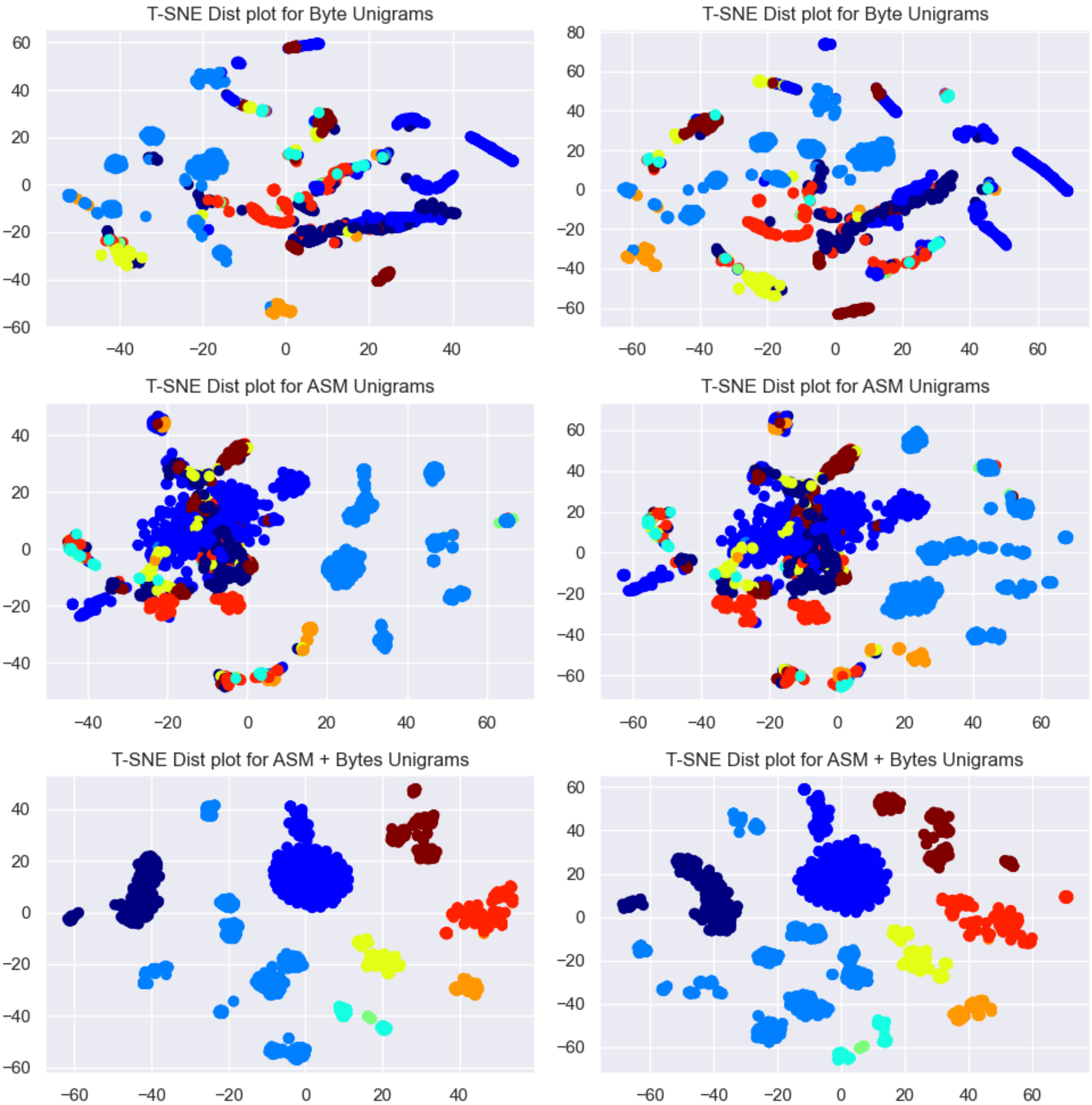


ASM Files Comparison



❖ TSNE Plots with perplexity {30 , 50}



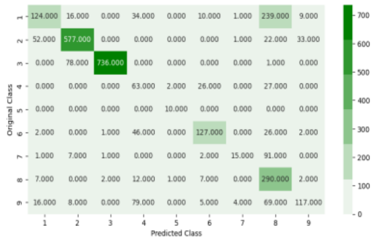
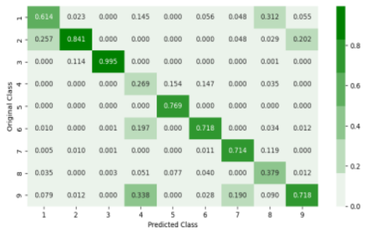
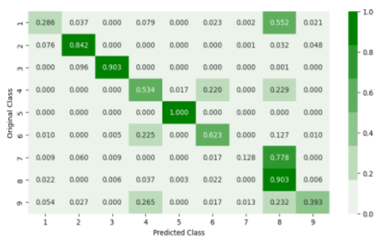
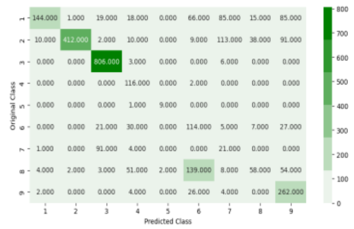
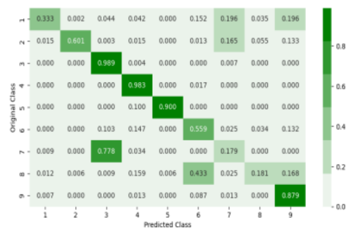
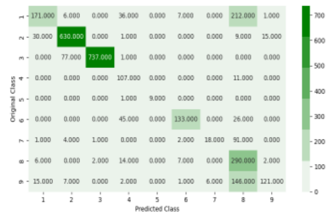
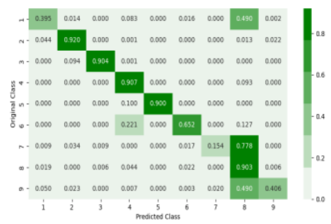
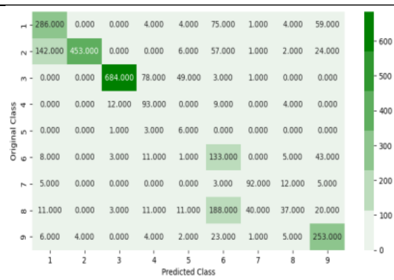
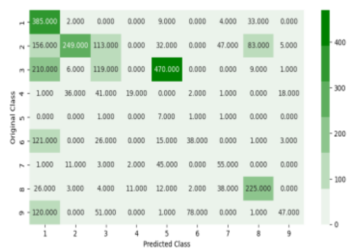
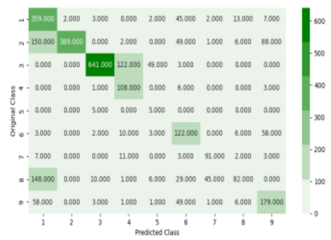


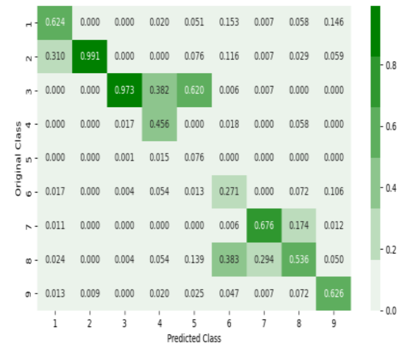
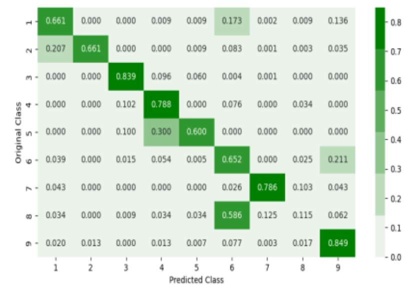
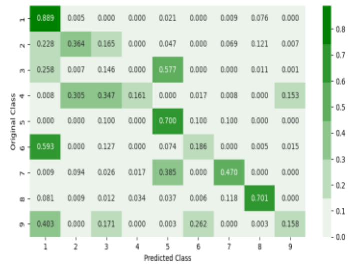
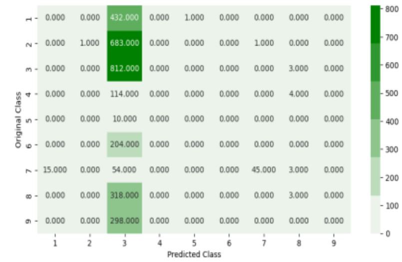
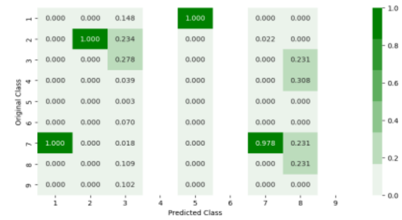
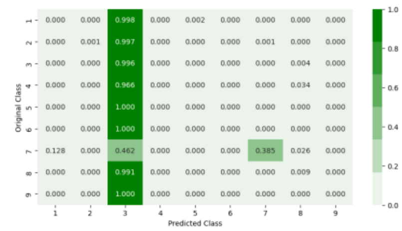
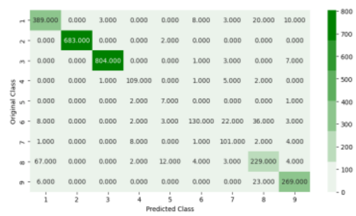
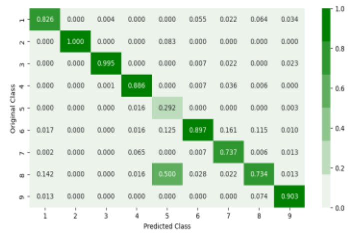
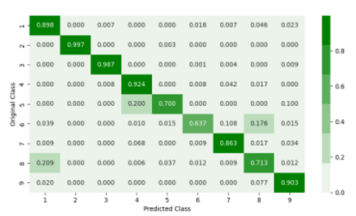
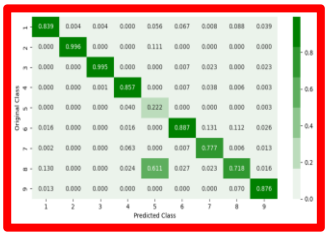
❖ Observations

1. Probabilities across various byte and assembly features exhibit a tendency toward uniform distributions within each class.

2. T-SNE Analysis highlights exceptional class segregation efficiency when considering the unigram features derived from the combination of byte and assembly code.

Model Training and Testing

Types of model /Feature Type	Byte Unigram	Asm Unigram	Byte + ASM Unigram
Gaussian Naïve Bayes	**Conf - Matrix**  **Precision - Matrix**  **Recall – Matrix** 	  	  
Multinomial Naïve Bayes			

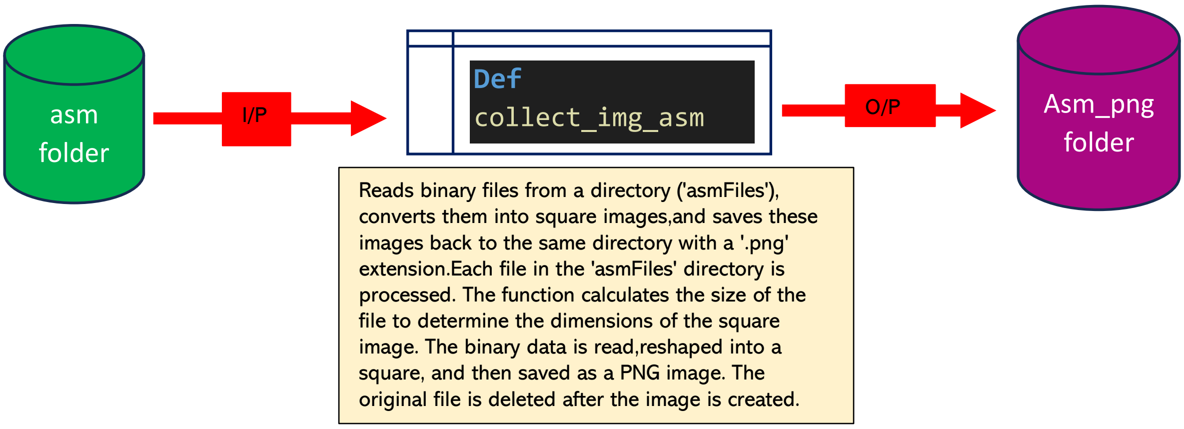
	 	 	 
Bernoulli Naïve Bayes	  	  	  

❖ Observation

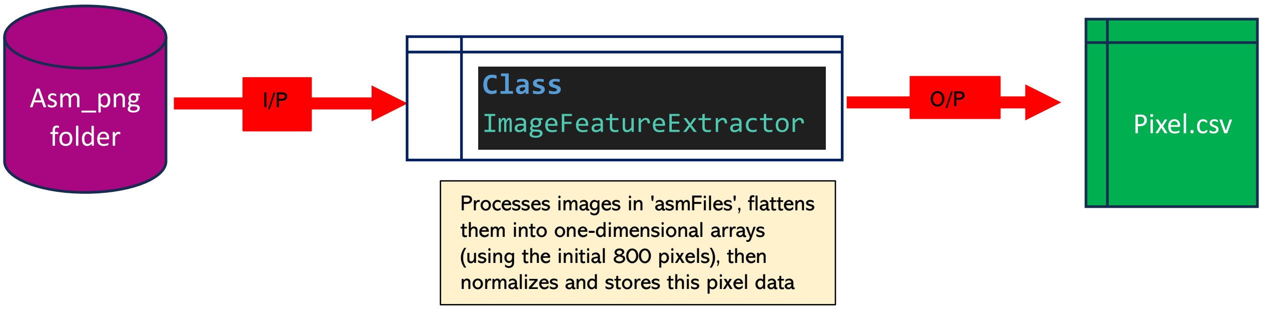
1. The analysis aligns with expectations, showcasing the standout performance of Bernoulli Naïve Bayes in accurately classifying diverse types of malware.

2. Despite achieving outstanding class segregation through featurization, precision for class 5 remained notably low due to the dataset's high imbalance, resulting in minimal precision for this specific class.

Introducing 'Visibility' Features



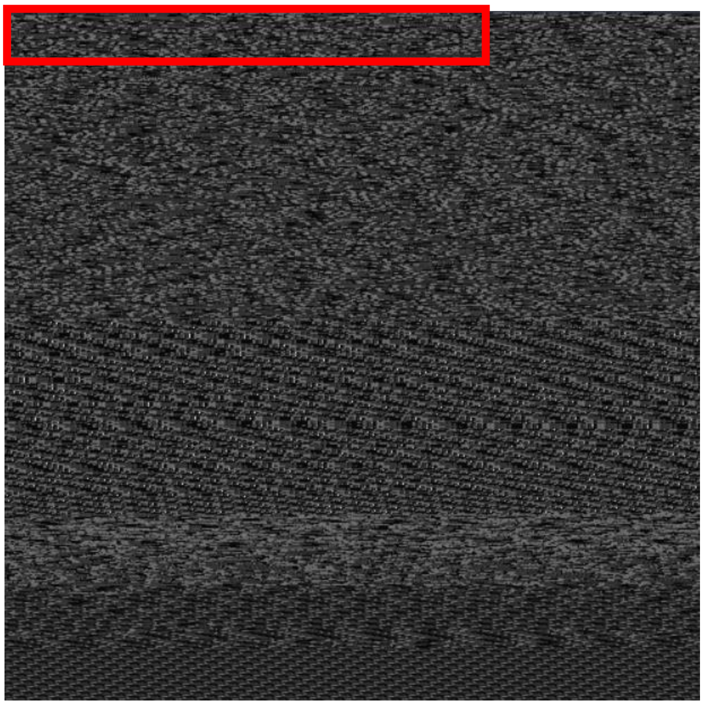
The function collect_img_asm() processes binary files from the 'asmFiles' directory, transforming them into square images and storing them back into the same directory as PNG files. For each file in 'asmFiles', it determines the file's size to calculate the dimensions of the square image. Then, it reads the binary data, reshapes it into a square format, and converts it to a PNG image. After this conversion, the original binary file is deleted to optimize storage. This process iterates through each file in the directory, ensuring that each binary file is converted into a corresponding square image, enabling easier visualization and potential downstream analysis.



The ImageFeatureExtractor class offers a static method to derive features from images, storing them as a CSV file. It operates by processing images within a specified directory, flattening each image into a one-dimensional array. These arrays are normalized, forming a feature dataset saved as a CSV file. The process reads images from the 'asmFiles' directory, flattens them (limited to the first 800 pixels), and aggregates these pixel values into a dataset. This dataset is normalized, organized into a DataFrame, and saved to a specified CSV file location. This method facilitates the extraction and storage of image features for further analysis or machine learning applications.

Analysis on Pixel Features

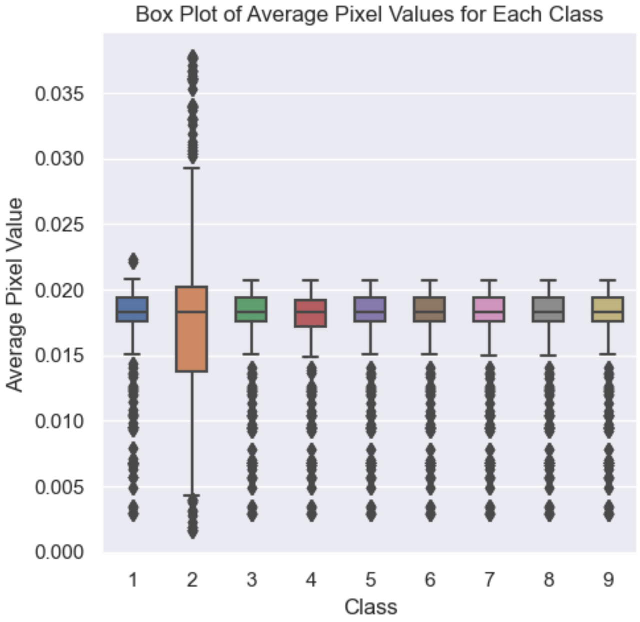
❖ Example of .asm -> .png

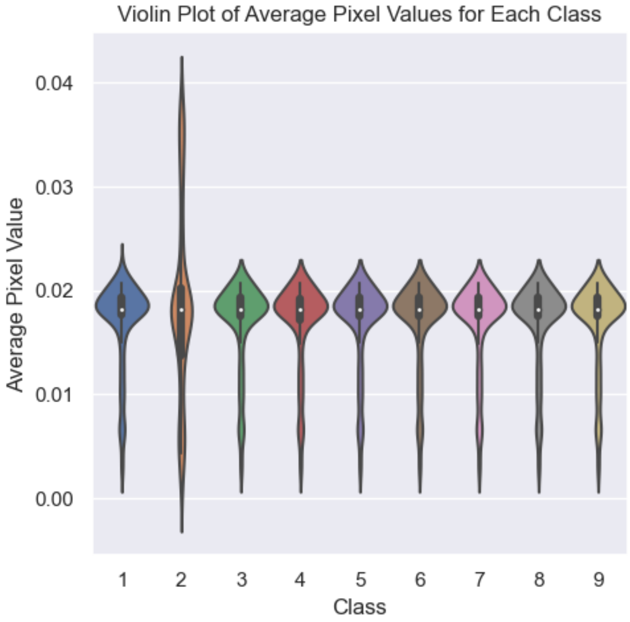


Description

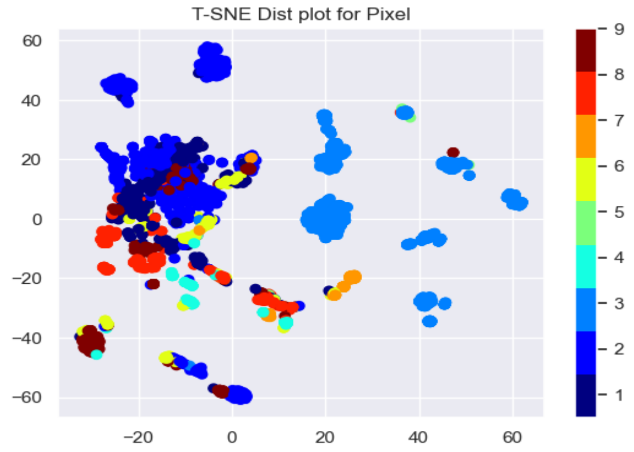
By extracting 800 - pixel values, this process captures basic visual information from images. These features might encapsulate patterns, textures, or distinctive elements within the images, aiding in subsequent analysis or pattern recognition.

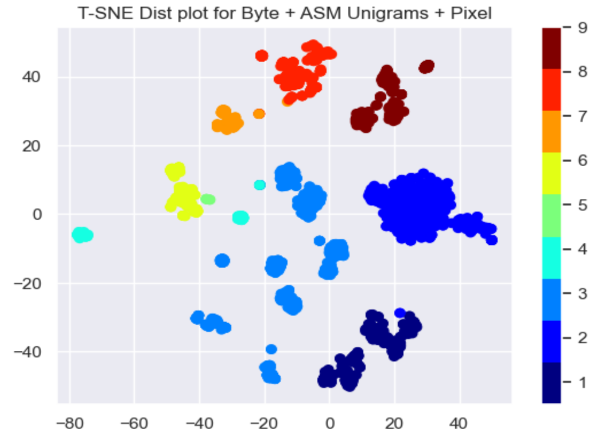
❖ Distribution related to each class





❖ T-SNE Segmentation



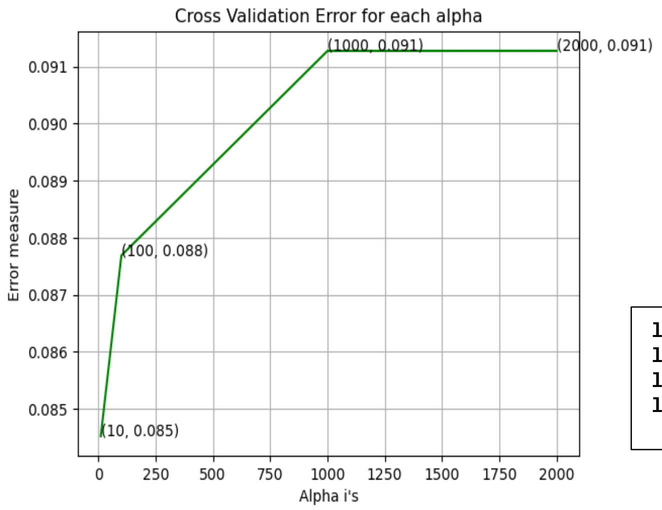


❖ Observations

1. Individual pixels don't provide effective segregation, yet combining Byte, ASM, and Pixel data results in exceptional segregation.

Model Building Using Caliberated ('sigmoid') XGB – Classifier

❖ Tuning number of estimators with error measure is equal to Log – loss



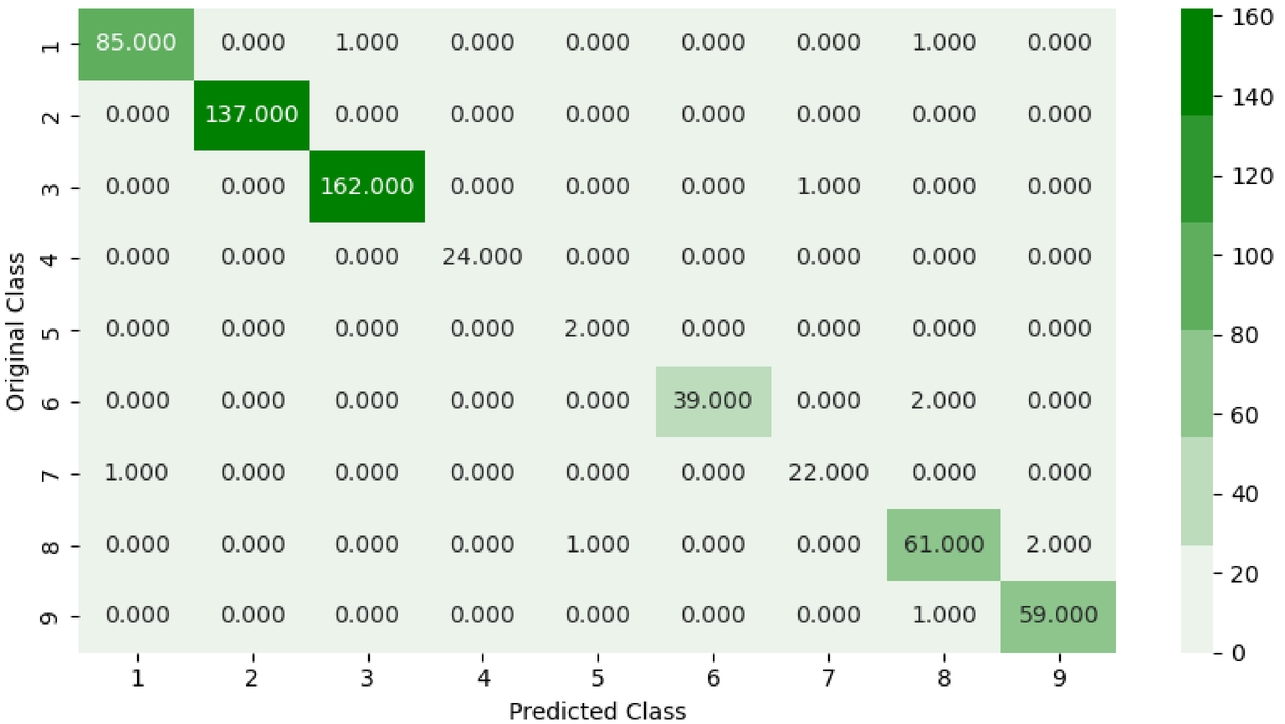

```
log_loss for c =  10 is 0.08452043784791066
log_loss for c =  100 is 0.087687436239066
log_loss for c =  1000 is 0.09127287903445112
log_loss for c =  2000 is 0.09127457459016397
```

The logarithmic loss, at varying values of 'c', exhibits a gradual increase: c=10 yields the lowest log_loss of 0.085, rising marginally at c=100 (0.088), and further at c=1000 and c=2000, plateauing around 0.091.

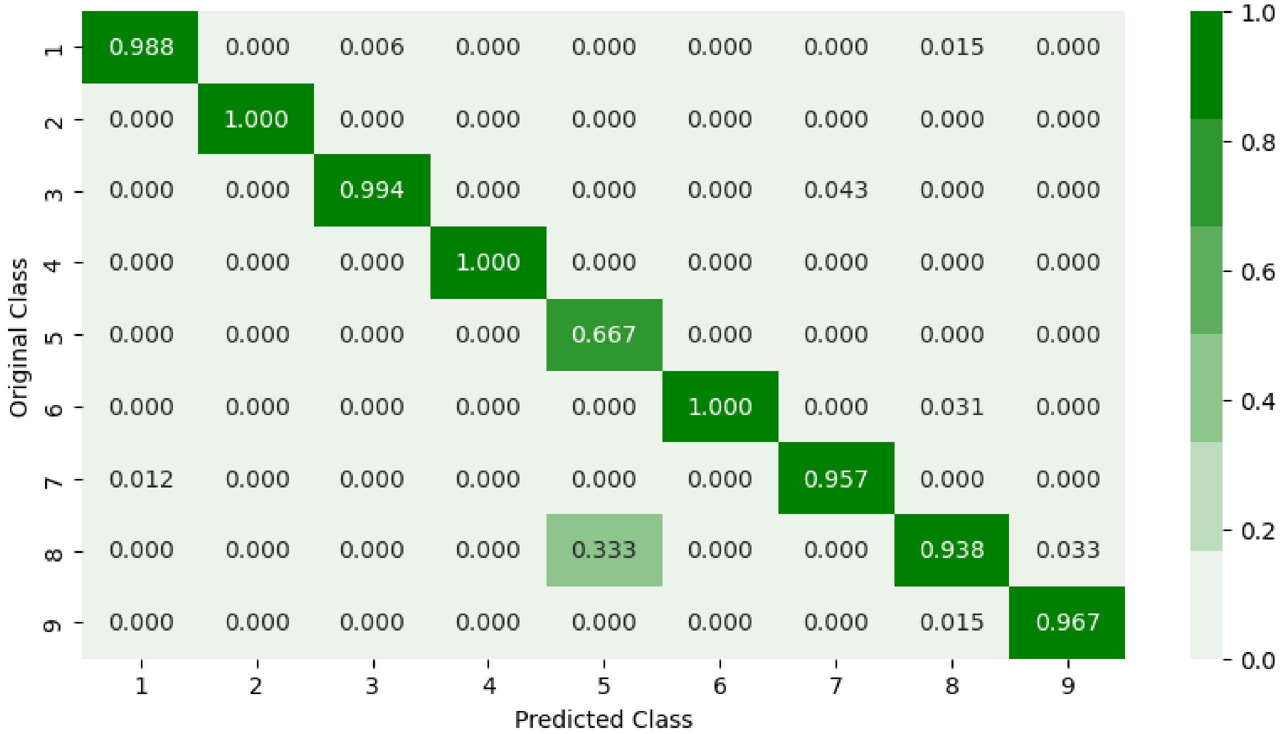
❖ Testing for Caliberated XGB – Classifier

```
For values of best alpha =  10 The train log loss is: 0.03863292153915008
For values of best alpha =  10 The cross validation log loss is: 0.08452043784791066

Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.98        87
           1       1.00      1.00      1.00       137
           2       0.99      0.99      0.99       163
           3       1.00      1.00      1.00        24
           4       0.67      1.00      0.80         2
           5       1.00      0.95      0.97        41
           6       0.96      0.96      0.96        23
           7       0.94      0.95      0.95        64
           8       0.97      0.98      0.98        60

    accuracy                           0.98       601
   macro avg       0.95      0.98      0.96       601
weighted avg       0.98      0.98      0.98       601

Multiclass Log Loss: 0.08452043784791066
Number of misclassified points  1.6638935108153077
```

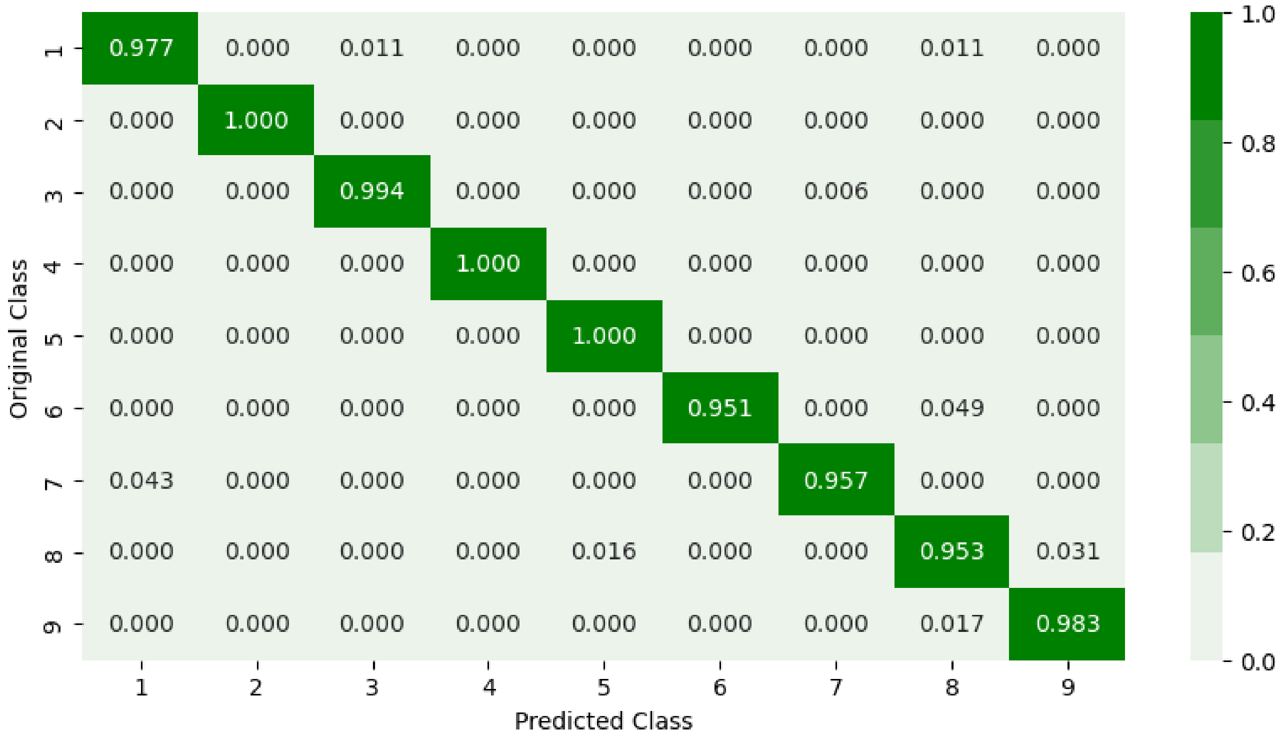
-- **Confusion matrix** --



Precision matrix



Recall matrix



Conclusion

The methodology navigates the labyrinth of malware detection through a meticulous, multi-stage process. Initiated by data preprocessing, it segregates files, processes byte and assembly files to extract pivotal features, and compiles a comprehensive DataFrame. The exploratory phase involves in-depth visualizations, examining class distributions, comparing byte and assembly features, and leveraging TSNE plots for nuanced insights. Model training unfolds, deploying diverse classifiers and underscoring the significance of amalgamating byte, assembly, and pixel data for refined classification. Further enrichment through 'visibility' features from images and pixel distribution analysis heightens comprehension.

The focal point emerges with the log loss analysis of a calibrated XGB classifier, shedding light on the significance of parameter tuning. The methodology concludes with rigorous model testing, unveiling precision and log loss metrics, offering a compelling portrayal of the model's proficiency in classifying diverse malware types.

Comprehensively, this methodology intertwines data exploration, feature extraction, model construction, and comprehensive evaluation. It stands as a testament to the convergence of cybersecurity and machine learning, offering a potent arsenal to combat the ever-evolving landscape of cyber threats. This holistic approach not only explores the diverse dimensions of malware detection but also embodies the collaborative synergy of cutting-edge technologies and rigorous methodologies, essential in fortifying digital defenses against persistent and evolving cyber threats.

References

- http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/
- https://arxiv.org/pdf/1511.04317.pdf
- First place solution in Kaggle competition: https://www.youtube.com/watch?v=VLQTRlLGz5Y
- https://github.com/dchad/malware-detection
- http://vizsec.org/files/2011/Nataraj.pdf
- https://www.dropbox.com/sh/gfqzv0ckgs4l1bf/AAB6EelnEjvvuQg2nu_pIB6ua?dl=0 " Cross validation is more trustworthy than domain knowledge."