

Minh Hung Le
Prof. Troia
Dec 10, 2022

1.Introduction

In this project we will extract the features of many malware families. In particular, we will prepare the data for training by using N-gram, TfidfVectorizer, Word2Vect, Bert, and their combinations. Then we will train the prepared dataset with four models including Random Forest, Decision Tree, K-nearest-neighbor and Support Vector Machine. We prefer them since these models employ a different classification strategy. As a result, we were able to identify the format which is most suitable for certain models and the format which is applicable for all models.

This paper will be organized as follows. In section 2, we will provide some background knowledge about each training model and the vectorization format we apply. Besides, we will discuss the advantages and disadvantages of various models and data formats. In section 3, we will explain the nature of the dataset that we will use for the experiment. Additionally, we will briefly introduce a few variables that we use in each training model. In Section 4, we will cover our experiment set up and the result of the experiment, and discussion of the results. Finally, In section 5 we will provide the conclusion, limitation of the project and plan for the future work.

2 Background

In this section, we will introduce the machine learning models and file formatting that we will use in this project.

2.1 Decision Tree (DT)

The decision tree is a tree-like structure similar to a flowchart. The nodes of the tree represent attributes, and the leaves represent classes. The advantage of decision trees is that they are generally easy to understand and construct. However, the decision tree appears to be overfitting since it builds the tree based on the set of given samples.

2.2 Random Forest (RF)

Random forest (RF) is a class of supervised machine learning techniques. Random forest models consist of a large number of individual decision trees. Each individual tree in the random forest model will operate and return the average output of the predictions. For classification problems, the prediction is the class that is selected by most of the trees. Random forest models generally perform better than decision trees and overcome the overfitting problem of decision trees.

2.3 K-nearest-neighbor (KNN)

The K-nearest-neighbor model is one of the simplest models for classification. Using the KNN model, samples are classified according to points that are similar to them. Thanks to its simplicity, KNN can produce predictions quickly when compared to other models. Indeed, the downside of this is that KNN models are prone to overfitting or wrongly classifying based on outliers.

2.4 Support vector machine (SVM)

The SVM model optimally separates samples into categories by using the hyperplane (n-1) when based on the sample dimension is n . Of course, we can have many different hyperplanes to separate many groups of samples. The advantage of SVM is that it works very well in high-dimensional space samples. However, with a large set of data SVM generally consumes a lot of time to train the model.

2.5 N-gram

The N-gram is the n-character slice of a long string into multiple strings. Rather than giving predictions based on the entire corpus, the N-gram model predicts based on slices of n words. Below we will demonstrate how a string will be sliced using N-gram format with the example of the string “Please give me good grade.”

N-gram	N-gram Generated List	Number of N-gram features
Unigram (1-gram)	“Please”, “give” , “me”, “good” , “grade”	5
Bigram (2-Gram)	“Please give”, “give me” , “me good” , “good grade”	4
Trigram (3-Gram)	“Please give me”, “give me good” , “me good grade”	3
Unigram + Bigram	“Please”, “give” , “me”, “good” , “grade”, “Please give”, “give me” , “me good” , “good grade”	9

N-gram models have the advantage of extracting many features from datasets. Nevertheless, the semantic significance of those features is limited to a few words close by. Therefore, N-gram models often fail to capture long distance meaning from a corpus.

2.6 TfidfVectorizer

TfidfVectorizer will transform text into feature vectors that can be used as input to the estimator. Each word will get a feature index. Tf-IDF transforms a word into a vector by comparing the number of times that word appears in a document with the number of documents the words appear in. Specifically, the equalization for vectorization of Tf-IDF is provided in the image below.

TF-IDF

TF-IDF is a measure of originality of a word by comparing the number of times a word appears in a doc with the number of docs the word appears in.

$$\text{TF-IDF} = \text{TF}(t, d) \times \text{IDF}(t)$$

Term frequency

Number of times term t appears in a doc, d

Inverse document frequency

$\log \frac{1 + \overset{\text{\# of documents}}{n}}{1 + \underset{\text{Document frequency of the term } t}{df(d, t)}}$

2.7 Word2vec

Word2vec is an algorithm for transforming the corpus of a word into a vector. The idea behind word2vec is that a training set of word vectors then places similar words close to each other in that space. In the same way, the words "food" and "meal" are often used interchangeably. They will therefore have very nearly identical vector representations. As a result, if we let Word2vec read the whole corpus, it would be able to group similar words near each other.

2.8 Bert

Bert learning framework for natural language processing (NLP). Bert is a deep learning model wherein every output element is attached to every input element, and the weightings among them are dynamically determined primarily by their connection. Thus, it is a transformer encoder that could read the corpus left to right and right to left. Then it will learn the context of each word based on all of its surrounding words(both left and right of the words). As a result, the semantic meaning and function of each instruction in the file will be much clearer.

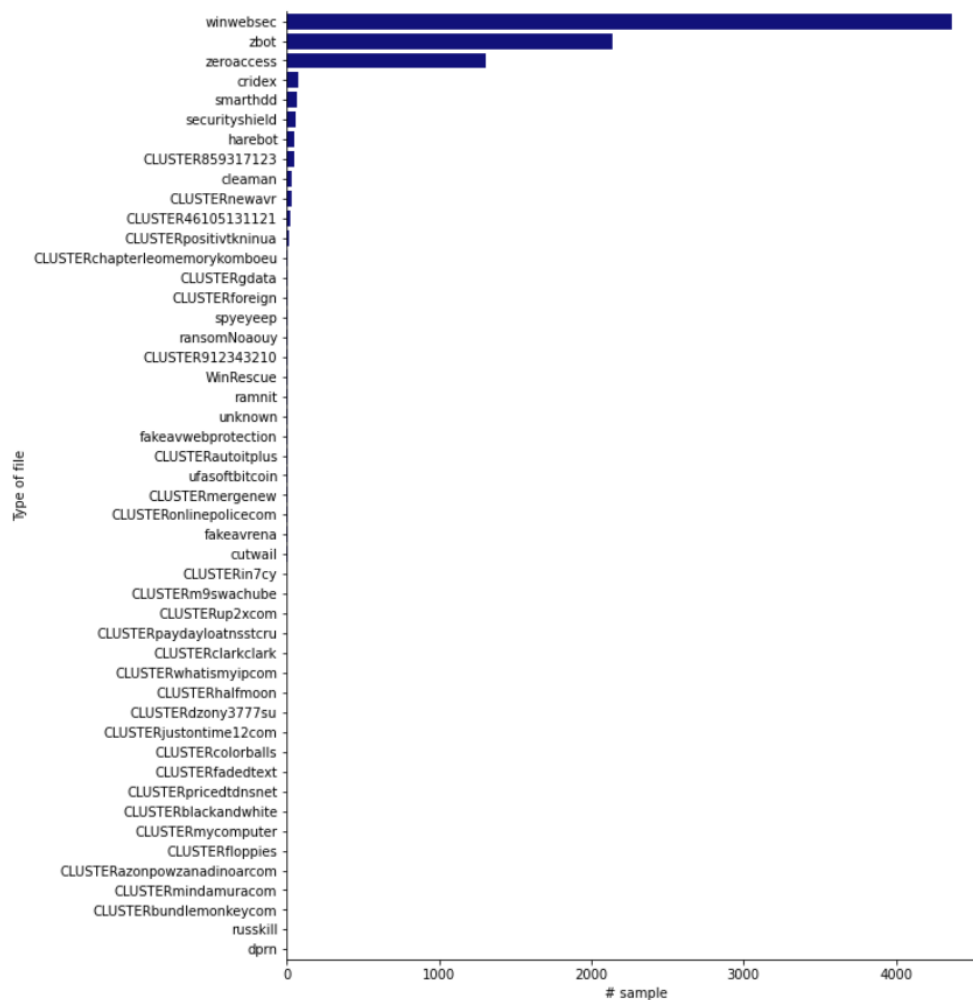
3 Dataset and Experimental Design

3.1 Dataset

The data set is archived at the link below:

https://mega.nz/file/vBlQjRpD#j53kyGVT6nQpclP0CSFQ9s_w3KCcrjXCyw-DFkrCBD4

The dataset contains a text file that contains assembly instructions for malware families. The names of each file and their ratio will be shown in the figure below.



In the data set above we could recognize many famous malware families such as:

Winwebsec: this is a trojan that fakes itself as an anti-malware program. Those malware files often attempt to output fake malware messages which require the user to pay money for removing the virus.

Zbot: this is the name of the specific Trojan virus that targets Windows computers with the intent of stealing sensitive financial information.

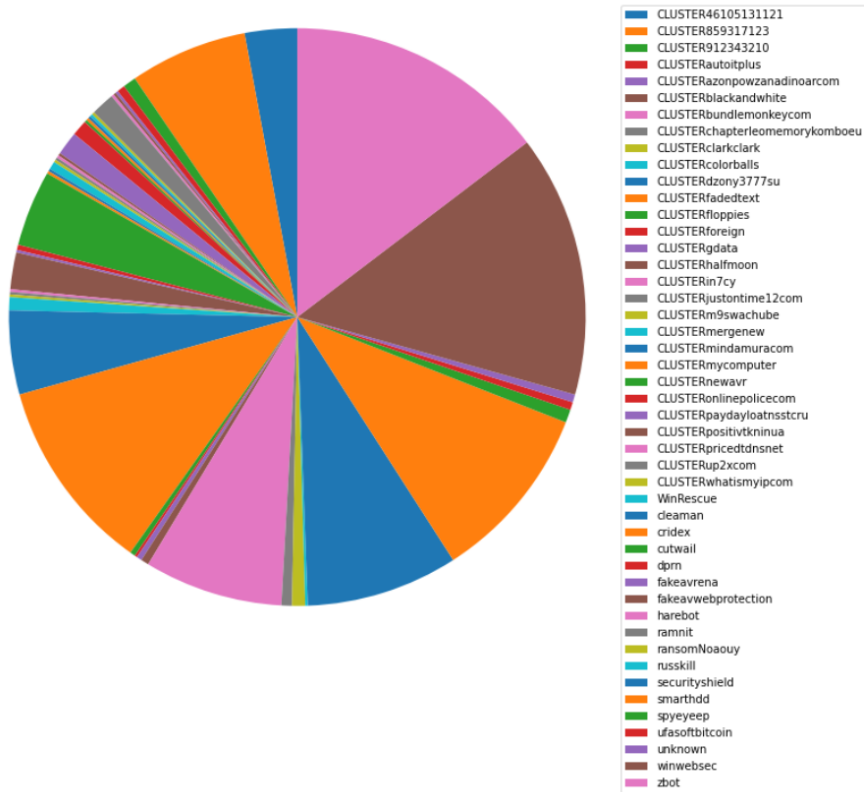
Zeroaccess: is a type of malware that infects Microsoft Windows operating systems. It is normally used to download other malware to the host system.

Cridex: a type of banking malware that can infect infected systems with bank credentials and other personal information in order to gain access to a user's financial records on the infected machines.

Smart HDD: is a fake system optimizer program. It attempts to scare computer users into believing that their computer hardware has issues that may lead to potential data loss

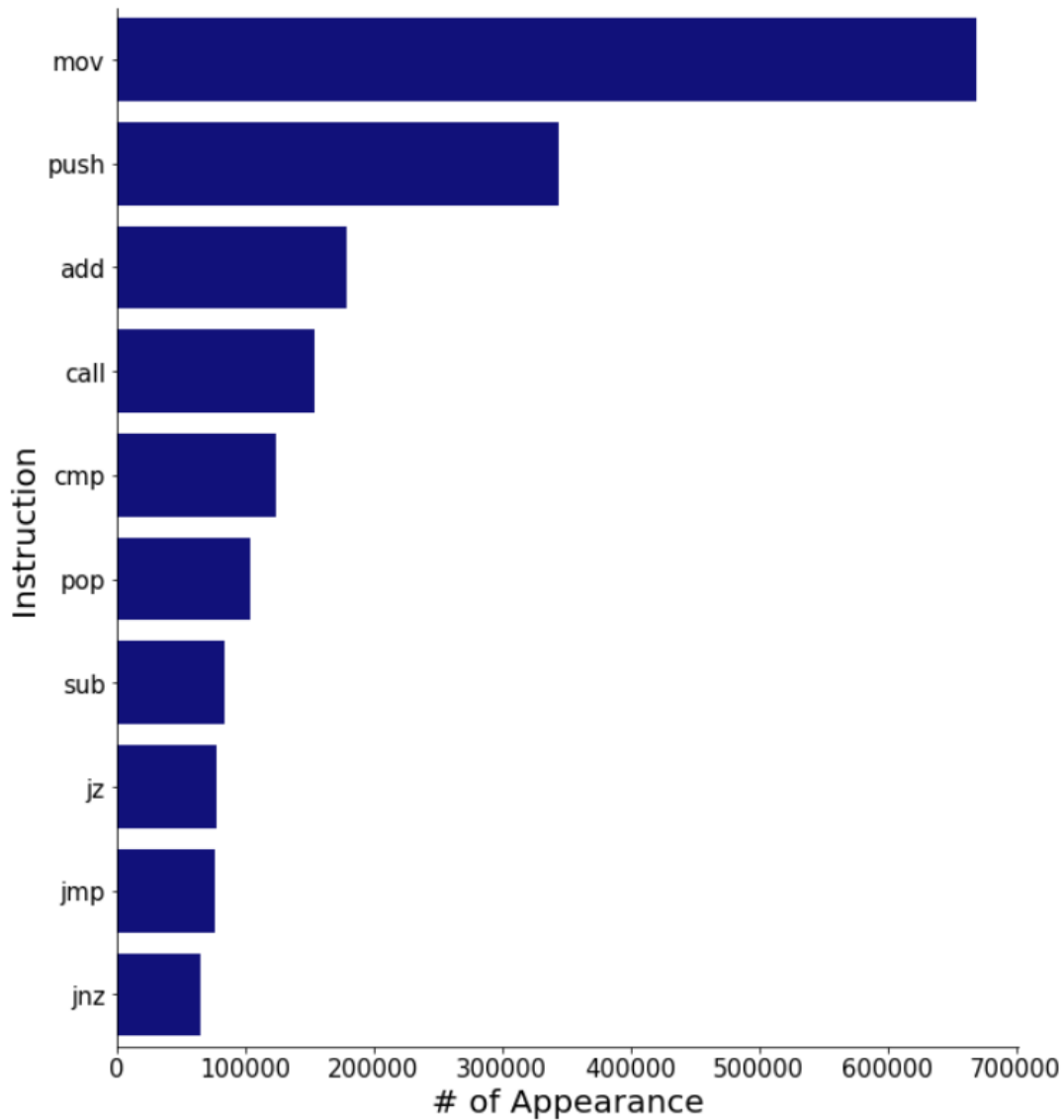
HareBot: is known to be a Trojan horse that allows remote attackers to gain access to the infected computer by using keystrokes from remote computers

From the data above we can see that winwebsec, zbot, and zeroaccess files have more than 1000 files. Indeed, using that unbalanced amount of data in training the model would overfit the model since our models would classify other files as those three files and get high accuracy. Therefore, I decided to randomly pick 100 files from the winwebsec, zbot, and zeroaccess categories while keeping the amount of files for other categories the same. Below is the ratio of each file after my modification above.



After the modification, we can see that after the modification of the number of files, the percentage of winwebsec, zbot, and zeroaccess files is still very high. In other words, we have an unbalanced data set. Therefore, we will train and optimize the model on the average F1 score for all experiments. F1 score is the balance between precision and recall. Thus, we can avoid biasing our models to predict minority families as the majority family.

Below is the appearance of the top 10 instructions that have a high frequency of appearance in all malware files. We also notice that some instructions appear more often than others. Thus, we doubt that there will be many repetitions of the top 10 instructions in each file.



3.2 Experimental Design

In each experiment, we will train and test the model with different vectorization formats for malware files. The specific technique for vectorization and pre-processing of the dataset will be discussed in bullet point 4 “Experiment and Results.” We will test each format over four models: Random Forest, Decision Tree, K-nearest-neighbor and Support Vector Machine. All models will be trained and tuned with GridsearchCV. The tuning hyperparameters is describe below:

+Random Forest:

'max_depth': [None,30,35,40]

'min_samples_split': [2,150,200]

+K-nearest-neighbor

'n_neighbors': [5,7,9,11,13,15,17,19,21]

+Decision Tree:

'max_depth': [None,4,6,7,8,30,32,35]
'min_samples_split': [2,3,4,5,35,10,16,20]

+Support Vector Machine:

'C': [0.1, 1, 10]
'kernel': ['poly']
'degree': [2, 5, 10]

Also, the training and test set will be split into the ratio of 80% for training and 20% for testing.

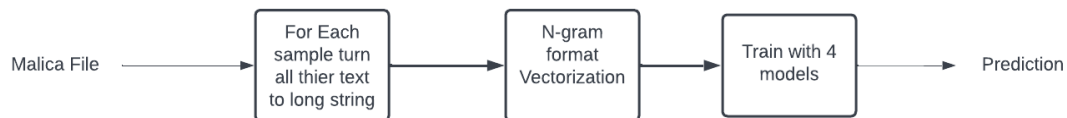
4 Experiments and Results

In this section, we will introduce the vectorization technique for each format, and we will give the testing results for each model. In the end we will analyze and compare the results of all vectorization techniques.


4.1 N-gram Format

For the N-gram vectorization, we use the similar set up vectorization technique from <https://www.kaggle.com/code/leekahwin/text-classification-using-n-gram-0-8-f1>.


The procedure for this experiment is:



Below is the dimension of the data set after we vectorize it with the N-gram technique. In "Data Dimension," the first number represents the number of samples we have. The second number describes the vector dimension of each sample.

N-Gram Feature Vector Data Dimension			
0	unigram	(682, 346)	
1	unigram_bigram	(682, 12245)	
2	bigram	(682, 11899)	
3	bigram_trigram	(682, 71380)	
4	trigram	(682, 59481)	

Below is the result for testing with N-gram format:

	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score	
0	unigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.556010	0.795620	0.490365	
1	unigram_bigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.534505	0.824818	0.483905	
2	bigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.551901	0.824818	0.499634	
3	bigram_trigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.575581	0.810219	0.483384	
4	trigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.584334	0.839416	0.495345	
5	unigram	KNN	KNeighborsClassifier()	0.424273	0.737226	0.376116	
6	unigram_bigram	KNN	KNeighborsClassifier()	0.427754	0.737226	0.411281	
7	bigram	KNN	KNeighborsClassifier()	0.458790	0.759124	0.417095	
8	bigram_trigram	KNN	KNeighborsClassifier()	0.464136	0.751825	0.376584	
9	trigram	KNN	KNeighborsClassifier()	0.448394	0.773723	0.439334	
10	unigram	Decision Tree	DecisionTreeClassifier(min_samples_split=3, ra...	0.480879	0.700730	0.375446	
11	unigram_bigram	Decision Tree	DecisionTreeClassifier(min_samples_split=3, ra...	0.482898	0.773723	0.415157	
12	bigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.496058	0.781022	0.404679	
13	bigram_trigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.503265	0.788321	0.448259	
14	trigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.476334	0.802920	0.388574	
15	unigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.122145	0.313869	0.173467	
16	unigram_bigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.170279	0.328467	0.179873	
17	bigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.173313	0.321168	0.173723	
18	bigram_trigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.162685	0.321168	0.176413	
19	trigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.132693	0.321168	0.184013	

4.2 TfidfVectorizer format:

Using TfidfVectorizer format is straightforward for this experiment since all sample text will be vectorized by TfidfVectorizer from the Sklearn library. Below is the process for experimentation with the TfidfVectorizer format.

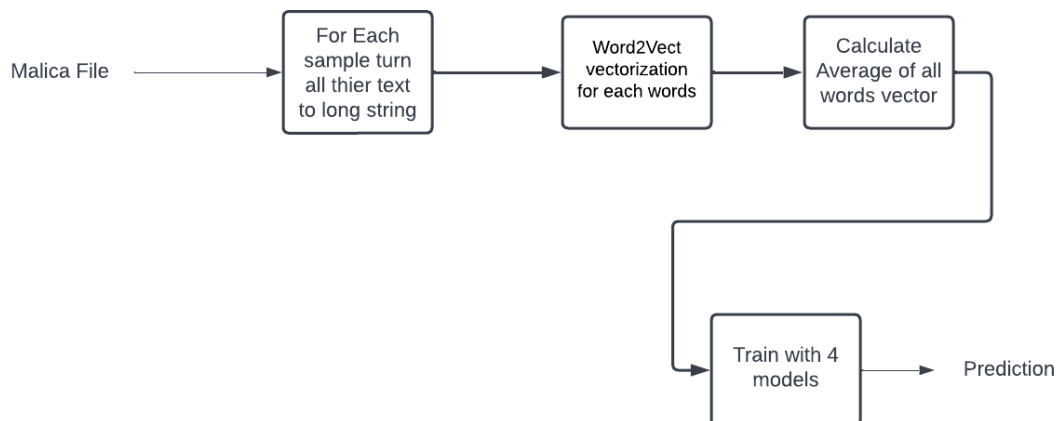


Below is the result for TfIdfVectorizer format.

	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score
0	TfidfVectorizer	Random Forest	(DecisionTreeClassifier(max_depth=35, max_feat...	0.555197	0.861314	0.583259
1	TfidfVectorizer	KNN	KNeighborsClassifier()	0.395784	0.810219	0.485617
2	TfidfVectorizer	Decision Tree	DecisionTreeClassifier(random_state=42)	0.417589	0.781022	0.417796
3	TfidfVectorizer	SVM	SVC(C=10, degree=10, kernel='poly')	0.518851	0.832117	0.549561

4.3 Word2Vect

With Word2Vec, we choose a vectorization dimension of 300 features. In other words, each sample will have 300 features. The Word2Vect function of the "gensim" library, however, implements vectorizing each word into a vector with 300 features. For example, Word2Vec will create 100 vectors with 300 features from a file with 100 words. Thus, we have to take additional steps to calculate the average vector for each sample, so that this average vector can represent the whole file. As a result, we follow the procedure below for this experiment:

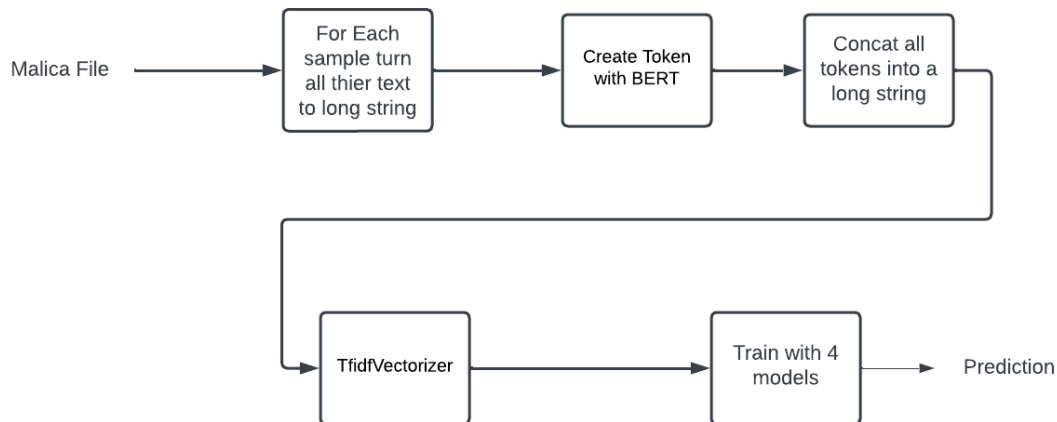


Below is Word2Vect experiment result:

	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score
0	Word2Vect	Random Forest	(DecisionTreeClassifier(max_depth=35, max_feat...	0.535534	0.868613	0.578491
1	Word2Vect	KNN	KNeighborsClassifier()	0.392431	0.824818	0.474739
2	Word2Vect	Decision Tree	DecisionTreeClassifier(max_depth=35, min_sampl...	0.424346	0.781022	0.424128
3	Word2Vect	SVM	SVC(C=10, degree=2, kernel='poly')	0.398977	0.788321	0.516126

4.4 Bert + Tfidf

Bert vectorization technique will transform a corpus of words into multiple tokens. Thus, each sample will have a different amount of tokens after the transformation. Therefore, we have to take additional steps to concat all the tokens into a long string. Then we vectorize this long string with TfidfVectorizer, so that all samples will have the same amount of features.

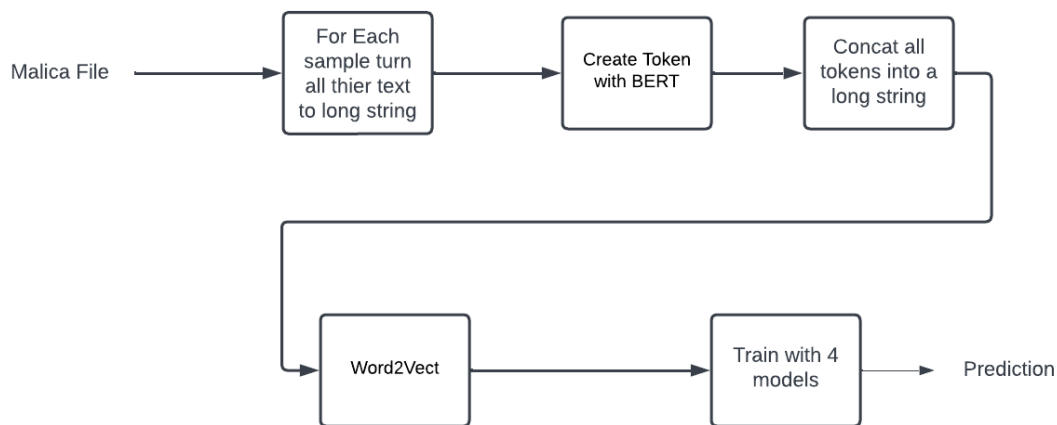


Below is Bert + Tfidf experiment result.

	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score
0	Bert Tdif	Random Forest	(DecisionTreeClassifier(max_depth=30, max_feat...	0.556233	0.861314	0.577888
1	Bert Tdif	KNN	KNeighborsClassifier(n_neighbors=7)	0.389121	0.810219	0.455840
2	Bert Tdif	Decision Tree	DecisionTreeClassifier(min_samples_split=5, ra...	0.438342	0.737226	0.353098
3	Bert Tdif	SVM	SVC(C=10, degree=10, kernel='poly')	0.495032	0.861314	0.560184

4.5 Bert + Word2Vect

Bert vectorization technique will transform a corpus of words into multiple tokens. Thus, each sample will have a different amount of tokens after the transformation. Therefore, we have to take additional steps to concat all the tokens into a long string. Then we vectorize this long string with Word2Vect, so that all samples will have the same amount of features. Word2Vect will have 300 dimensions.

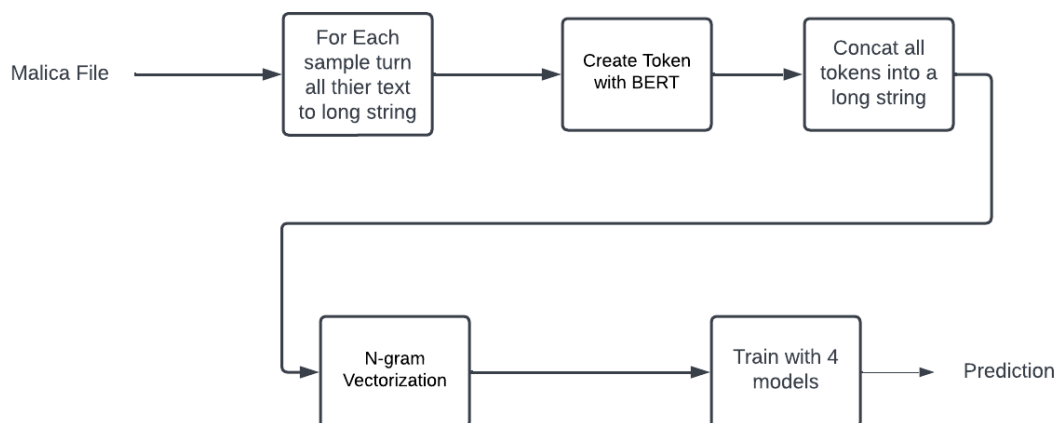


Below is Bert + Word2Vect experiment result:

	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score
0	Bert Word2Vect	Random Forest	(DecisionTreeClassifier(max_depth=35, max_feat...	0.519832	0.854015	0.558384
1	Bert Word2Vect	KNN	KNeighborsClassifier()	0.395417	0.810219	0.465145
2	Bert Word2Vect	Decision Tree	DecisionTreeClassifier(min_samples_split=3, ra...	0.400501	0.788321	0.422045
3	Bert Word2Vect	SVM	SVC(C=10, degree=2, kernel='poly')	0.386777	0.781022	0.467797

4.6 Bert + N-gram

Bert vectorization technique will transform a corpus of words into multiple tokens. Thus, each sample will have a different amount of tokens after the transformation. Therefore, we have to take additional steps to concat all the tokens into a long string. Then we vectorize this long string with N-gram, so that all samples will have the same amount of features.



Below is the dimension of the data set after we vectorize all tokens with N-gram technique.

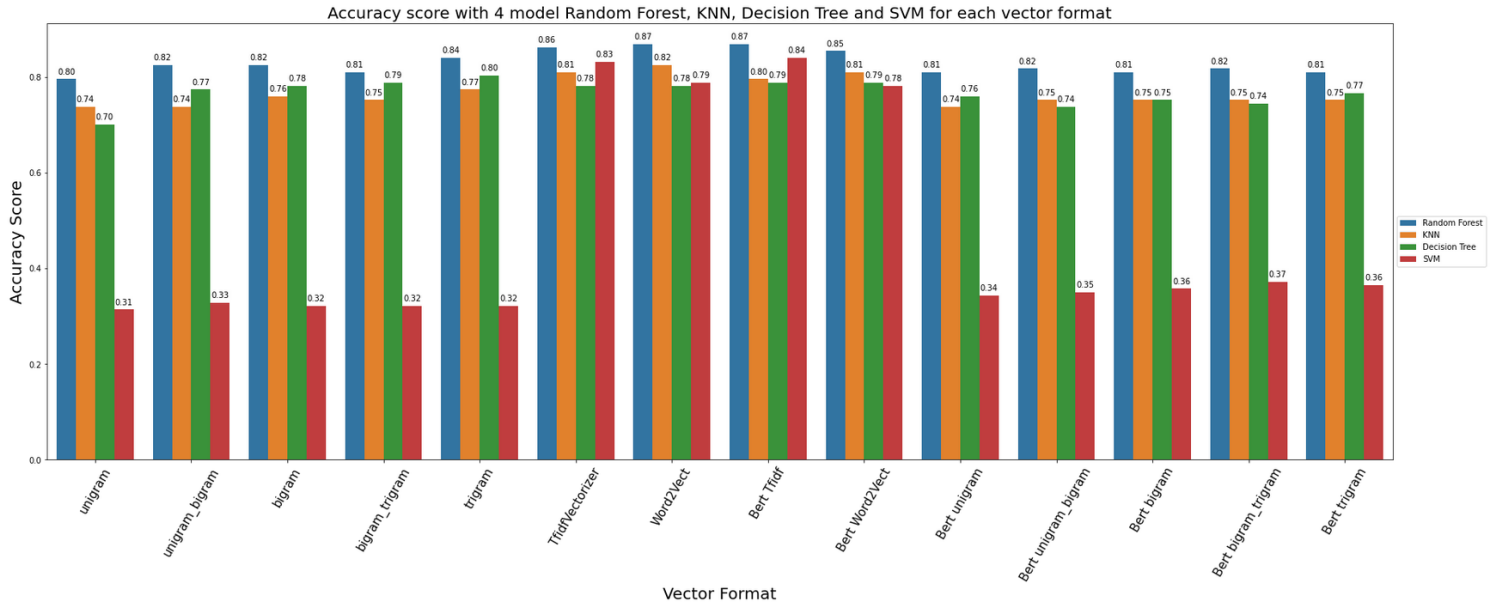
	N-Gram Feature Vector	Data Dimension
0	unigram	(682, 216)
1	unigram_bigram	(682, 6674)
2	bigram	(682, 6458)
3	bigram_trigram	(682, 48850)
4	trigram	(682, 42392)

Below is Bert + N-gram experiment result.

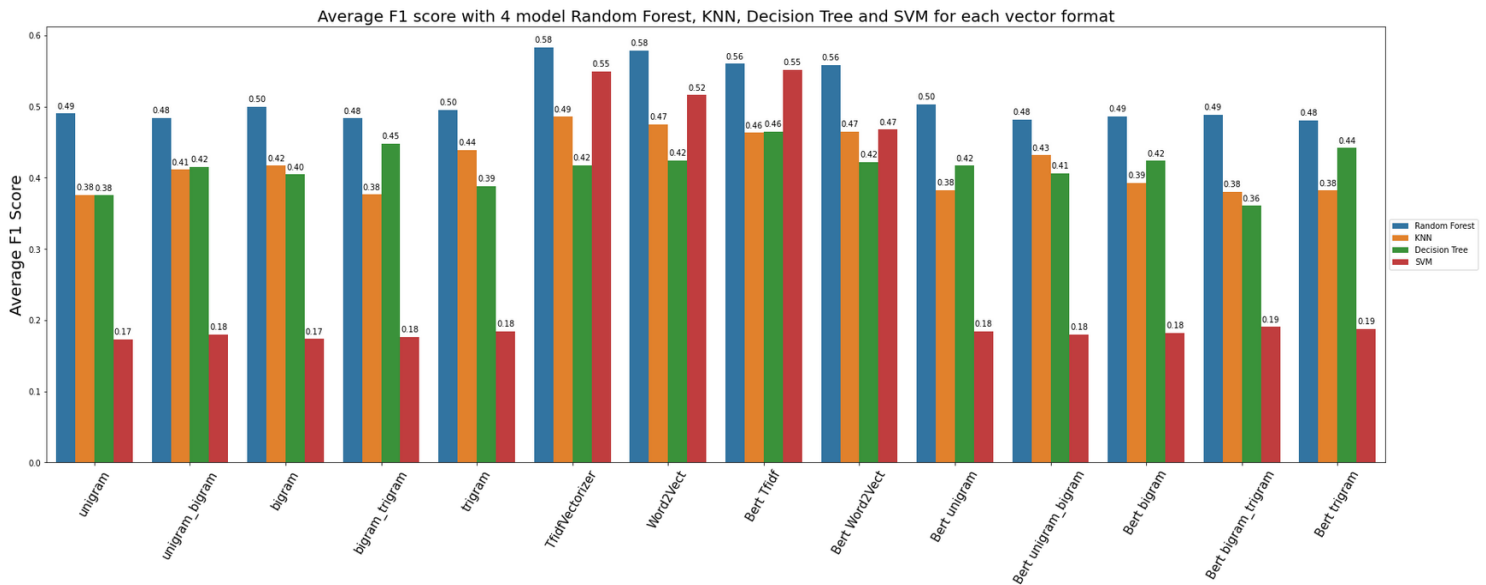
	Vector	Model	Calibrated Estimator	Validation Score	Test Accuracy Score	F1 Score
0	Bert unigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.609401	0.810219	0.502937
1	Bert unigram_bigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.601841	0.817518	0.482033
2	Bert bigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.569435	0.810219	0.486182
3	Bert bigram_trigram	Random Forest	(DecisionTreeClassifier(max_features='auto', r...	0.570692	0.817518	0.488205
4	Bert trigram	Random Forest	(DecisionTreeClassifier(max_depth=30, max_feat...	0.561790	0.810219	0.480309
5	Bert unigram	KNN	KNeighborsClassifier()	0.439760	0.737226	0.382571
6	Bert unigram_bigram	KNN	KNeighborsClassifier()	0.439915	0.751825	0.431667
7	Bert bigram	KNN	KNeighborsClassifier()	0.456715	0.751825	0.392533
8	Bert bigram_trigram	KNN	KNeighborsClassifier()	0.497875	0.751825	0.380355
9	Bert trigram	KNN	KNeighborsClassifier()	0.471232	0.751825	0.382244
10	Bert unigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.476643	0.759124	0.417053
11	Bert unigram_bigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.499432	0.737226	0.406662
12	Bert bigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.486800	0.751825	0.423834
13	Bert bigram_trigram	Decision Tree	DecisionTreeClassifier(min_samples_split=4, ra...	0.486153	0.744526	0.360531
14	Bert trigram	Decision Tree	DecisionTreeClassifier(random_state=8888)	0.484182	0.766423	0.441895
15	Bert unigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.137056	0.343066	0.184367
16	Bert unigram_bigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.122326	0.350365	0.179719
17	Bert bigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.212816	0.357664	0.182356
18	Bert bigram_trigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.199150	0.372263	0.190513
19	Bert trigram	SVM	SVC(C=10, degree=2, kernel='poly')	0.181817	0.364964	0.187576

4.7 Result comparison:

The result of accuracy score for each model with different vector format:



The result of average f1 score for each model with different vector format:



In the experiment, the N-Gram format had the poorest performance of all formats. As mentioned in the background, the N-gram format captures about one to three instructions for each feature of the training vectors. Indeed, N-gram models will struggle to capture long-distance contexts. Of course, that will cause significant problems for classification between different malware families. Logically, many times code will declare a variable, but they will use that variable many lines later in the code. Of course, the N-gram format will fail to recognize those variables or instructions that refer to the same variable since it only captures one to three instructions at once. Besides, N-gram format will perform even worse if the programmer writes their code in Object-Oriented-Design format where functions and classes are declared to be reused in different places. N-gram format may not recognize that some functions are reused or where the end of

each chunk of functions is located. As a result, our model outputs the least accurate predictions in N-gram format.

TfidfVectorizer outperforms n-gram formats in terms of accuracy and f1 score. Based on the appearance of each word in all documents and the number of words appearing in a single document, TfidfVectorizer will capture and vectorize the corpus. We know different malware with different design purposes will need a certain amount of instructions compared to other instructions. To steal essential information in the system, some malware will require much more read instructions such as "la" and "li". In contrast, malware that aims to destroy the operating system may require more write instructions, such as "move". Therefore, by overviewing the appearance of each instruction in all documents and in each malware family, TfidfVectorizer could extract an appropriate feature for training models. Consequently, TfidfVectorizer achieves high scores for all models that we train.

Among all formats, Word2Vec also achieves high accuracy and F1 scores. This format observes all corpora and creates a vector by observing similarity between words. As a result, Word2Vec can group those words together. In fact, that feature is quite useful for our models to recognize different malware categories since Word2Vec models can detect some malware code behavior. For example, you could use Word2Vec to detect a branching instruction "je" that appears near an increment instruction "add", and help our model understand that it implies a loop. Of course, each malware will require a specific amount of loop and the content inside the loop will also help our model distinguish between the malware. Thus, Word2Vec vectorization techniques also work very well for classifying different malware families.

When it comes to Bert format, the Bert transformer will learn the context and the meaning of each instruction both forward and backward. Therefore, Bert format might have a better understanding of where a function or a loop starts and where it stops compared to other models. In the experiment, we only combined Bert with different vectorization formats such as TfidfVectorizer, Word2Vec, and N-gram. Below, we will discuss the results of those combinations.

The Bert format TfidfVectorizer is the most effective combination since it gives us the highest accuracy score and F1 score for all models. As discussed, we can see that the Bert tokenization provides semantic meaning and code structure to malware files. When combining those with TfidfVectorizer, we will also capture how much of each separate function appears in different malware families. As a result, our models can give a slightly more accurate prediction compared to TfidfVectorizer alone.

When combining BERT with Word2Vec, we combine the semantic meaning of the code structure with how those functions and instructions are likely to appear together. Unfortunately, it rarely improves performance when compared to Word2Vec alone. As we take the average of each

token vector, we might lose or underestimate the importance of some tokens generated by BERT. As a result, we rarely improve performance.

Finally, combining BERT and N-gram format rarely gives us better performance than N-gram format alone. When using N-gram to vectorize BERT generated tokens, we might lose the semantic meaning of each token generated by BERT format. Essentially, BERT format helps the model understand the semantic meaning and structure of the file, while N-gram format breaks those tokens into equal chunks. As a result, the BERT process is undermined. Thus, we still receive bad performance when combining BERT and N-Gram.

5. Conclusion and Future Work

We examined multiple different vectorization techniques to prepare the training data set in this paper. Additionally, we experiment with those formats with diverse machine learning models to avoid the situation where one model would favor one format over the other. From the experiment above, we can conclude that BERT + TfidfVectorizer will give us the most accurate results for malware classification. Specifically, this combination allows us to better understand the coding structure of malware code and the appearance of those structures. As a result, all training models achieved accuracy between 80 -85%. In contrast, the n-gram format would give us the most inaccurate accuracy score of below 80% for many models and below 40% for SVM models. Due to the reuse of many assembly instructions throughout the malware files, the features of different malware files in the N-gram format overlap. Due to this, it is very difficult for a SVM model to classify those families.

In the future, we would perform similar experiments with more formats such as ELMO and HMM2Vect. In this project, we are unable to set up experiments for these two formats due to a lack of hardware resources. In addition, there is not yet a HMM2Vect implementation for Google Colabs Notebook that we can follow and apply smoothly to this project. About ELMO format, it runs smoothly on tensorflow version 1.15. Unfortunately, there is zero way for us to downgrade our tensorflow version in Google Colabs notebook to match that requirement. Further, the RAM limitation of hardware also prevents us from trying more complicated models such as CNN or SVM with a radial kernel. Basically, our notebook crashed while we were training and fine tuning those models. Finally, the datasets we experiment with are unbalanced, since some malware families dominate others. Therefore, recording more samples of the minority malware family will be the next challenge for us. Therefore, in the extension of this project, we will improve our hardware and implement more formats as well as more complicated models with much more balanced data.

Citation

Hamdaoui, Y. (2021, March 24). *TF(Term frequency)-idf(inverse document frequency) from scratch in python* . Medium. Retrieved December 13, 2022, from <https://towardsdatascience.com/tf-term-frequency-idf-inverse-document-frequency-from-scratch-in-python-6c2b61b78558>