ABSTRACT :

JavaScript has played a crucial role in web development, making it a primary tool for hackers to launch assaults. Although malicious JavaScript detection methods are becoming increasingly effective, the existing methods based on feature matching or static word embeddings are difficult to detect different versions and obfuscation of JavaScript code. To solve this problem, we present a novel detection method that consists of adaptable context analysis and efficient key feature extraction. The key to our approach is context analysis based on dynamic word embeddings. Furthermore, as a classification module in the method, TextCNN can effectively extract key features.

INTRODUCTION:

Most websites use JavaScript to create dynamic interaction, which brings great convenience to people's daily life, but also brings many security problems. Malicious JavaScript codes are challenging to detect because they can transcode malicious code by obfuscation methods, such as logical operation, string segmentation and string compression. Furthermore, JavaScript version iteration also makes detection more difficult.

The existing methods for malicious JavaScript detection mainly rely on feature matching or static word embeddings. However, feature learning based on static word embeddings does not represent the real context information of code, which means the problem of polysemy in the code is not resolved, making undesirable detection effects. Besides, the analysis methods based on feature matching can only detect known malicious samples.

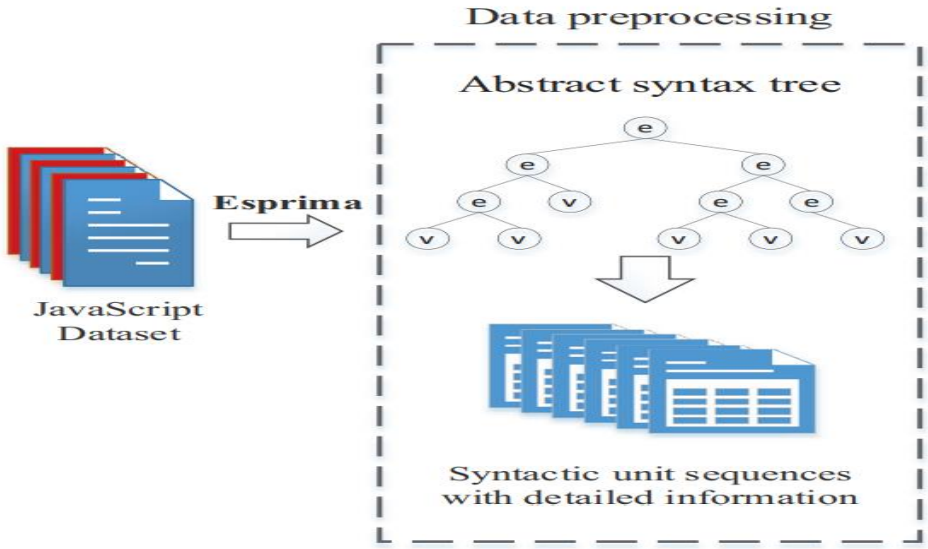
Aiming at the problem of undesirable detection effects caused by insufficient use of code information in existing methods, we present a novel detection method using adaptable context analysis and efficient key feature extraction. To achieve this, Word2Vec and two Bidirectional Long Short-Term Memory (Bi-LSTM) layers are introduced into feature learning of the model. It is feature learning based on dynamic word embeddings that makes full use of code information. In TextCNN,

the object of convolution operation is the word vector, and convolution kernel width is consistent with the dimension of word vectors, which can extract features effectively. Then we perform a features selection process using the Tabu Search algorithm in order to reduce the dimensions of dynamic word embedding and improve performance.

METHODOLOGY:

- **Data description and preprocessing:**

  We collected 4,000 non-attack JavaScript files and 4,000 XSS attack files, then rendered them as structured data. Then we deleted the URL's and the IP's from them. Then we performed a grammatical analysis process to obtain Syntactic unit sequences with detailed information. Finally we converted it to static word vectors.

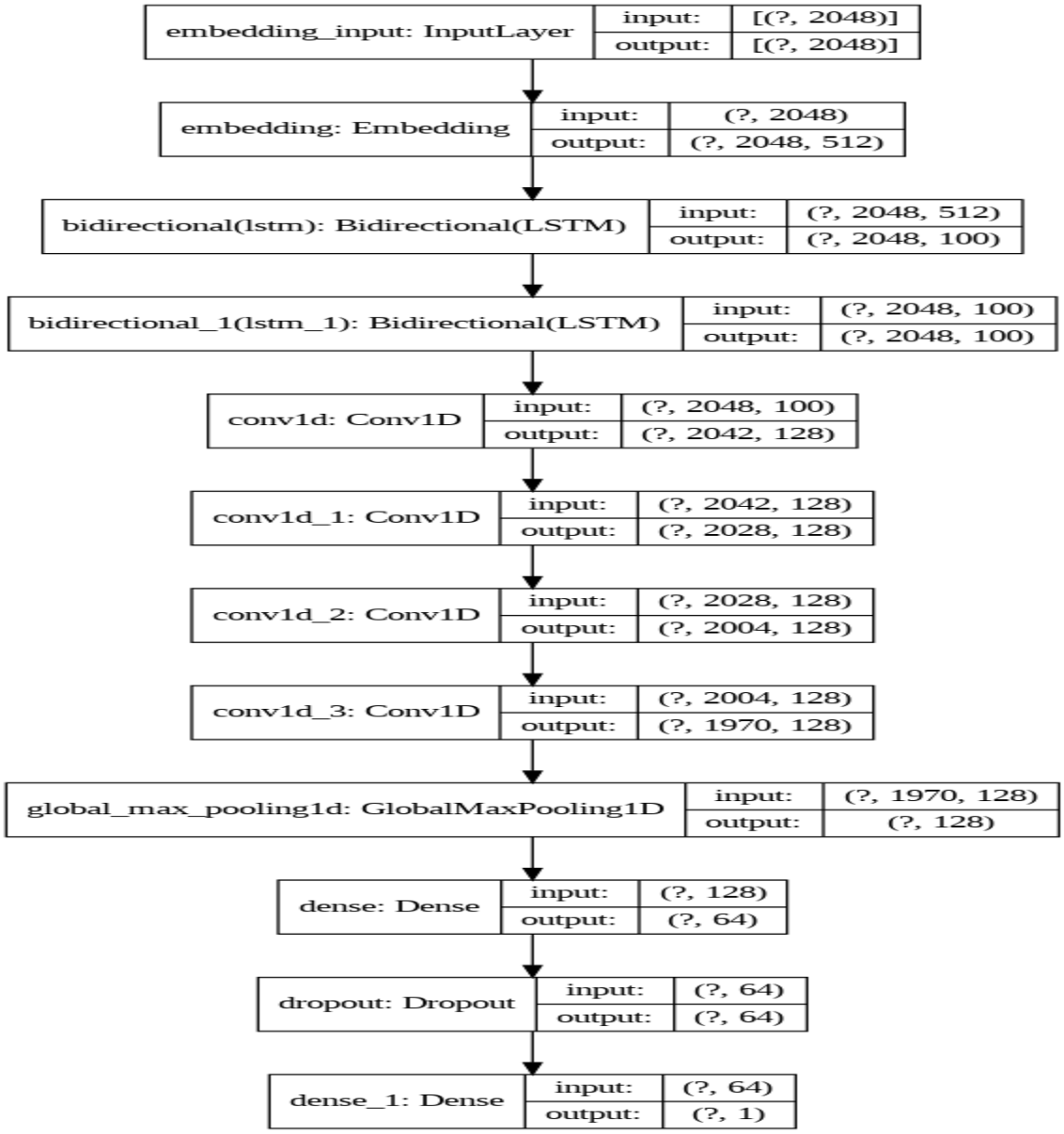


  We collected 4,000 payload files that contain an XSS attack and 4,000 files that do not contain an attack, then we decode them, then we perform the Generalization process, where we convert each number or series of numbers to the number 0 and each link to "http://u" and then we After that we did the segmentation process. Finally we converted it to static word vectors.

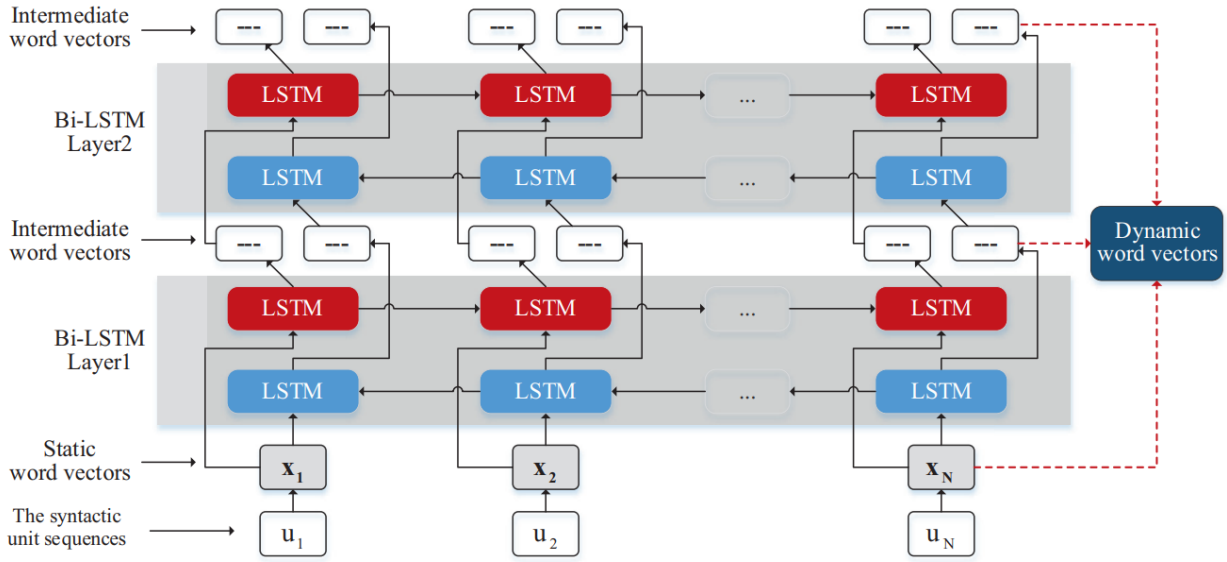
- **Extract dynamic word embedding:**

  At this point, we have trained a deep learning model to classify JavaScript files (and payload files), where 0 indicates that the file does not contain an attack

and 1 indicates that the file has an attack. The figure below shows the structure of the model. We give the Embedding layer the following parameters (vocab_size=500000, embedding_dim=512, input_length=2048) and for the Bidirectional layer (units=50, input_shape=(None, 50), return_sequences=True) for both layers. As for the four layers of Conv1D (filters=128, activation='tanh'), but they differ from each other in kernel_size, where we give the first layer 7, the second 15, the third 25, and the fourth 35. Finally the fullyConnectedLayer. for the first layer (units=64, activation ='relu') and for the output layer (units=1, activation='sigmoid')



Before training and after pre-treatment of files. We divided the 8000 files into 4 chunks, each chunk out of 2000 files, then for each chunk we divided it into 1600 files for training and 400 for testing, then we trained each chunk on the training data, then we extracted the hidden states from the two BiLstm layers and then we tested the model on the test data. Then we divided the model into two parts, the first starting from the input layer to the last BiLstm layer, and the second from the first Conv1D layer to the end, then we used the first model to extract the hidden states of the test data. Then we weighted sum between the hidden states of the two layers and the static word Embedding, where we weighed the last BiLstm layer with a value of 0.45, the previous layer with a value of 0.35, and the static word Embedding with a value of 0.20, so we have the dynamic word Embedding with a dimension of (2048,100) for each file.



- **Feature selection using tabu search algorithm:**

Tabu search (TS) is a metaheuristic search method employing local search methods used for mathematical optimization. It was created by Fred W. Glover in 1986 and formalized in 1989.

We used this algorithm after the failure of many feature selection algorithms, because the data we are working on is a hidden state extracted from a deep learning model, which means that spatial information is as important as temporal information, So we cluster the hidden states to 34 clusters. Since each

hidden state is from the dimension (2048,100), we do the selection process on the columns, that is, we will reduce the second dimension of each hidden state so that the new dimension is, for example, (2048,50) or (2048,60) or …

Algorithm:

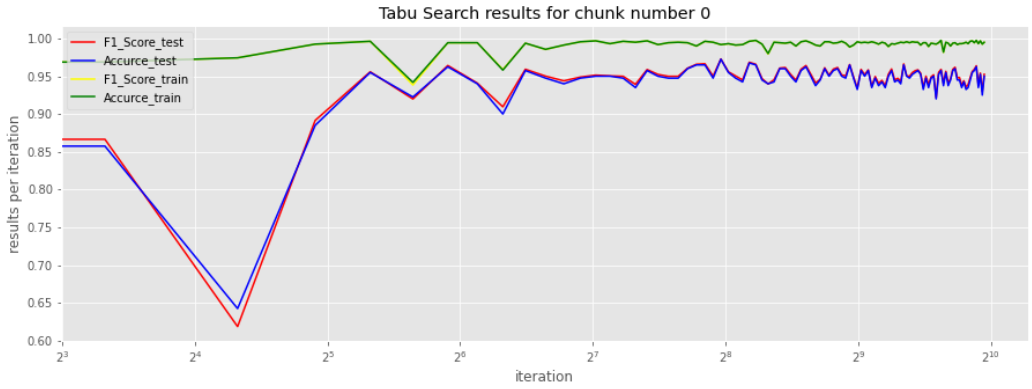
Input: Number_of_chunk=4, number_of_class=34, Tabu_tenur=100,

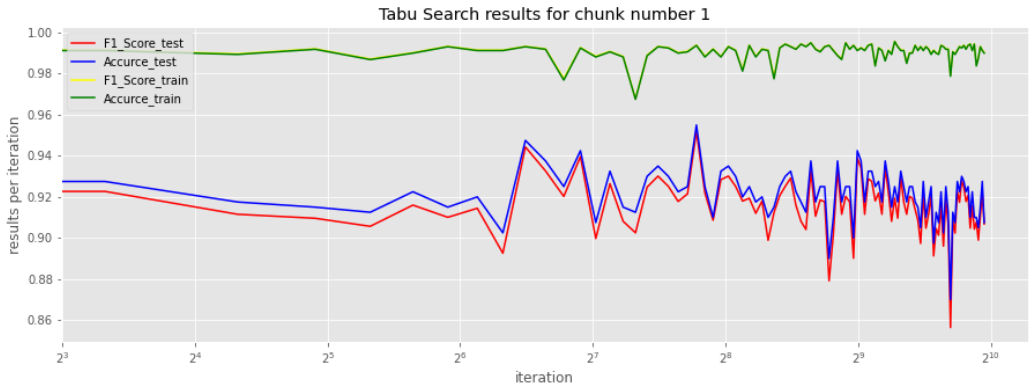
Output: solutions,F1_train,Ac_train,F1_test,Ac_test

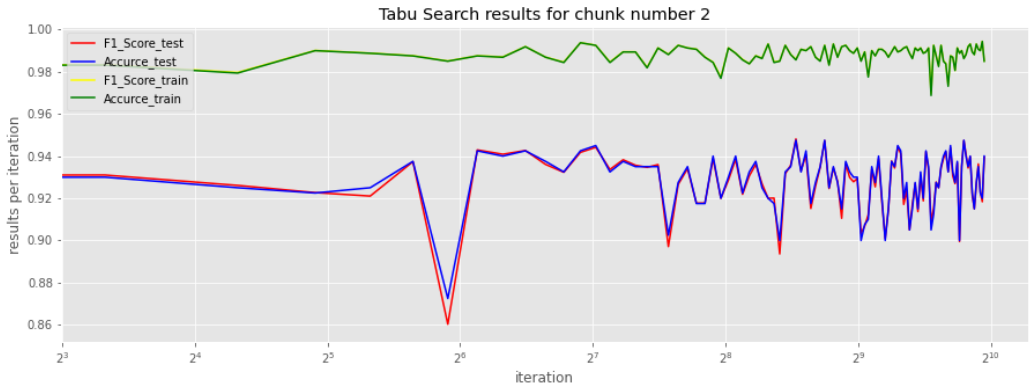
```
for i=0 = number_of_chunk do
 Load(X_train , y_train , X_test , y_test) for chunk number i
 number_of_iteration = X_train.shape[2]*10
 if i=0:
  model = create_model()
  model.save()
 else:
  Tabu_list = load(Tabu_list_i)
  agglomerativeClustering = load()
  model = load_model()
 end if
 X = get_statistical_information(X_train)
 cluster_train = agglomerativeClustering.fit_predict(X)
 X = get_statistical_information(X_test)
 cluster_test = agglomerativeClustering.predict(X)
 current_solution = np.random.randint(low=0,high=2,size=(number_of_class,X_train.shape[2]))
 solutions,F1_train,Ac_train,F1_test,Ac_test = EMPTY_LISTS
 for step=0 → number_of_iteration do
  x_train = apply_mask(X _train,current_solution,cluster_train)
  history = model.fit()
  x_test = apply_mask(x_test,current_solution,cluster_test)
  y_pre = model.predict(x_test)
  f1_test,ac_test = evaluate_result(y_pre,y_test)
  y_pre = model.predict(x_train)
  f1_train,ac_train = evaluate_result(y_pre,y_train)
  Tabu_list = update_tabu_list(current_solution)
  aspiration_criteria  = check_aspiration_criteria(f1_test,ac_test)
  if not aspiration_criteria:
   model = load_model()
  else:
    model.save()
  End if
  solutions.add(current_solution)
  F1_train.add(f1_train)
  F1_test.add(f1_test)
  Ac_train.add(ac_train)
  Ac_test.add(ac_test)
  save (solutions, F1_train, Ac_train, Ac_test)
  current_solution = generate_neighbour_solution(current_solution)
 End for
End for
```

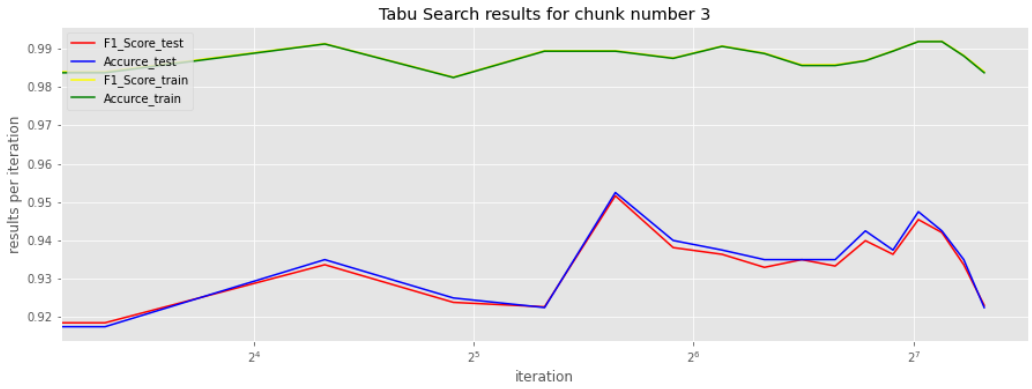
RESULTS:

For javascript file:

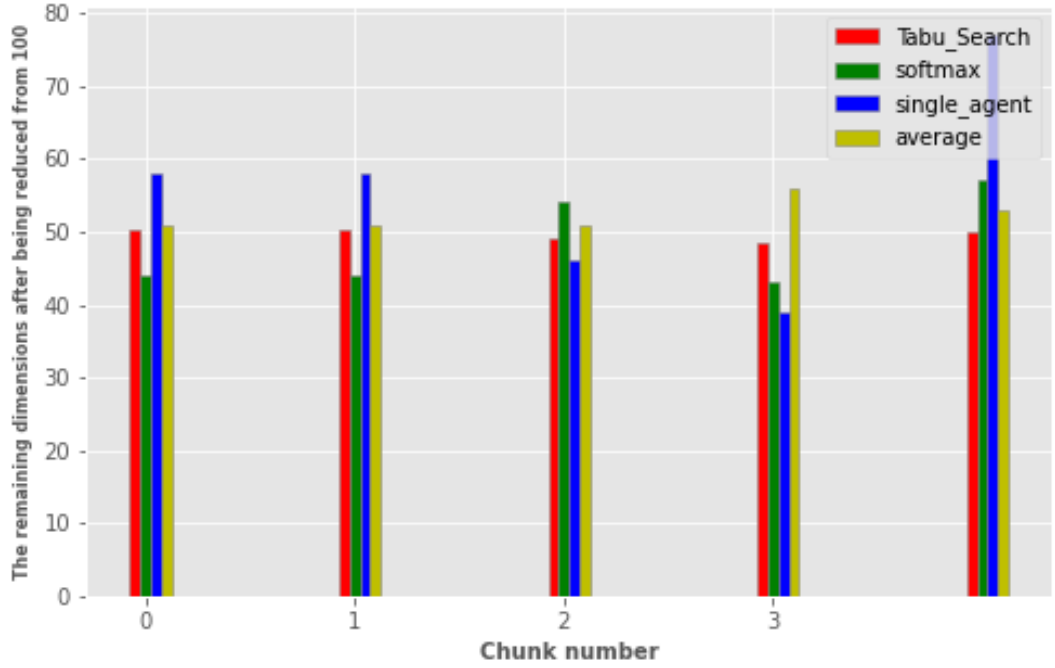


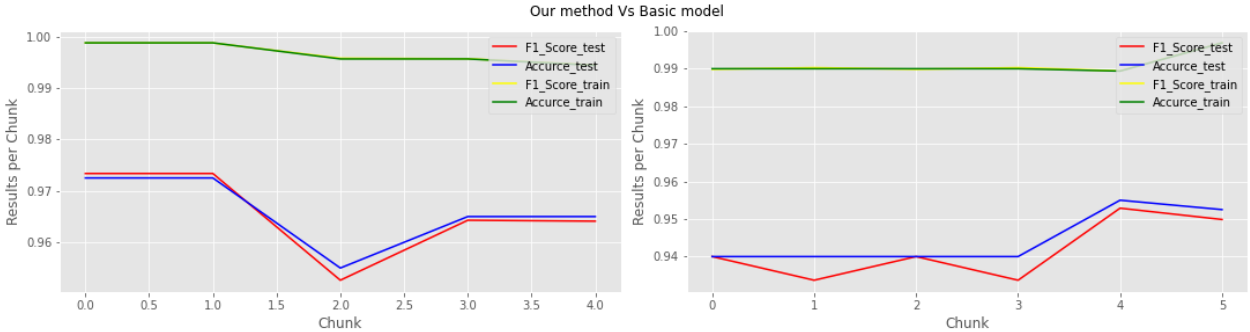


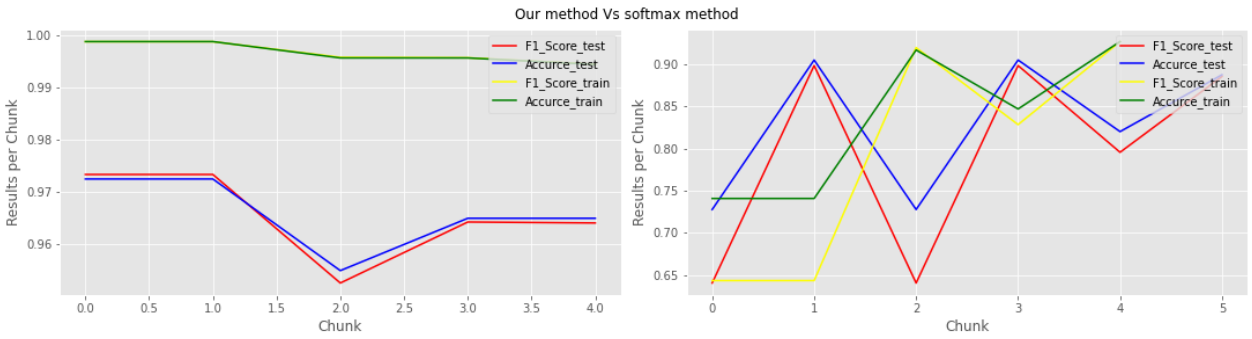




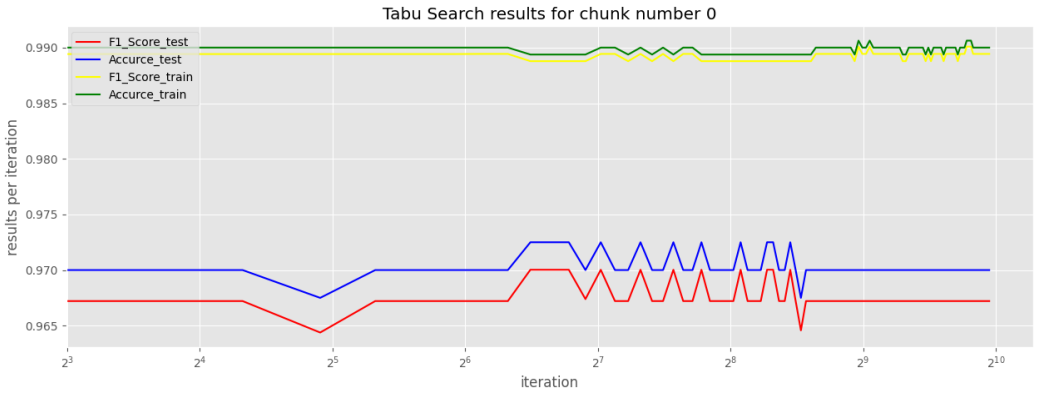
We compared our method with a method of selecting very powerful features based on reinforcement learning techniques in terms of the amount of dimension reduction and performance improvement, and the results were as follows:

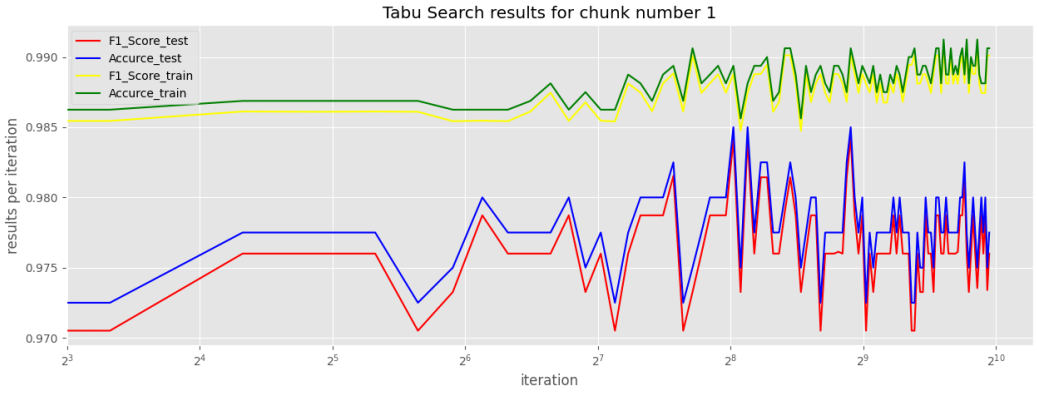


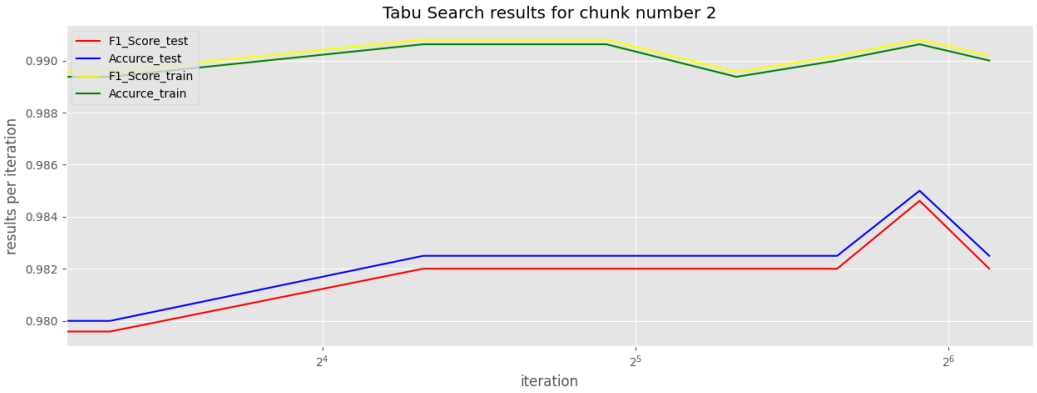




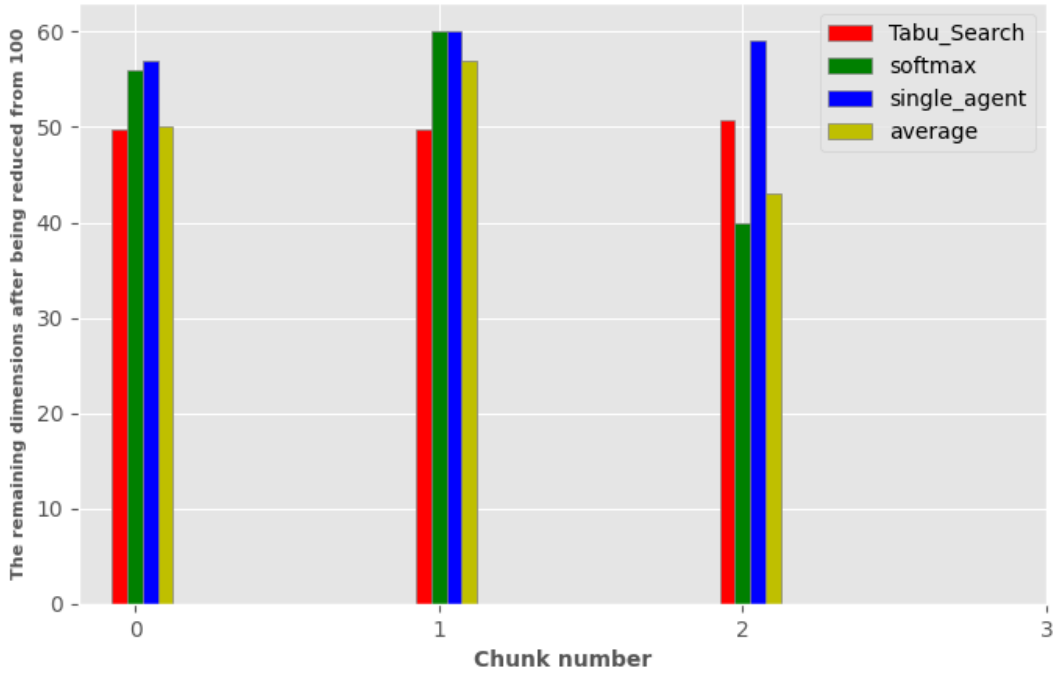
For payload file:



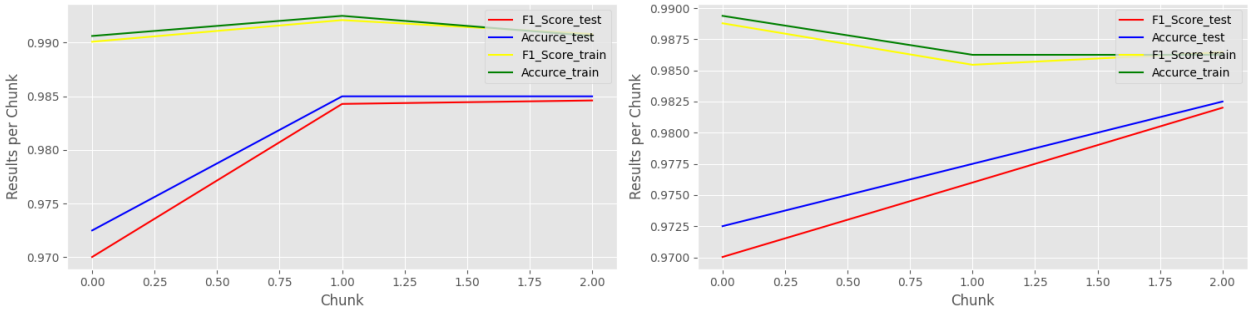




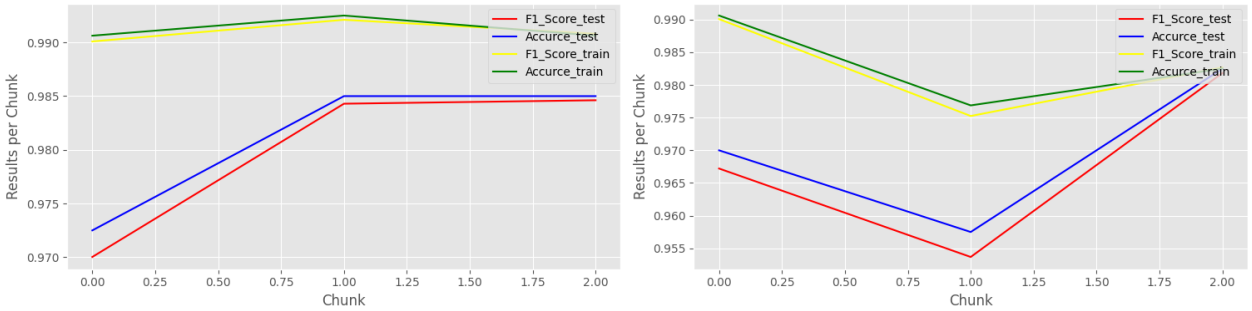
Also same comparison. we compared our method with a method of selecting very powerful features based on reinforcement learning techniques in terms of the amount of dimension reduction and performance improvement, and the results were as follows:



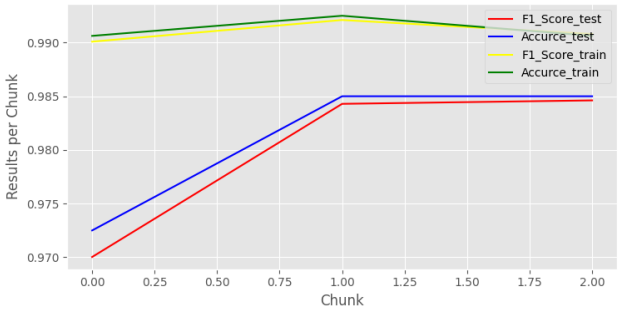
Our method Vs Basic model

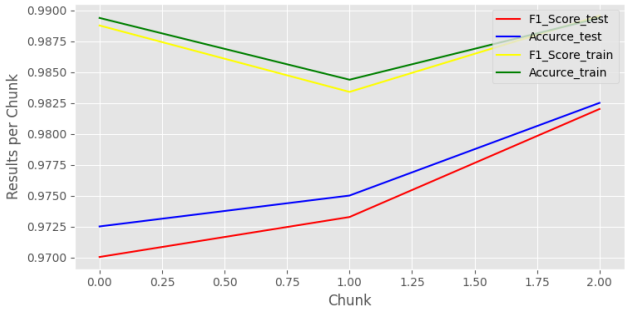


Our method Vs softmax method



Our method Vs single_agent method





Our method Vs average method