Revisiting The 2017 RE Data Challenge: Transformer-based Language Models for Security Requirement Identification

Ahmed Elrashidy
Department of Physics, Astronomy and Geosciences
Towson University
Towson, USA
aalras2@students.towson.edu

***Abstract*—Since the introduction of transformer based models in 2017, Transformer models revolutionized natural language processing (NLP) and became the most influential and widely adopted architectures for a variety of tasks, including machine translation, text summarization, language generation, and more. The Transformer's self-attention mechanism enables it to capture long-range dependencies in a sequence, making it highly effective in handling sequential data like text.**

In this paper I will revisit the 25^{th} IEEE International Requirements Engineering Conference Data Challenge. I will start by summarizing the papers submitted to IEEE journal and then explore using transformer-based language models for identifying security requirements. I will show the robustness of multiple transformer-based models and their flexibility using transfer learning.

***Index Terms*—Requirements Engineering, Language Models, Transfer Learning, Transformers**

I. Introduction

As software systems become increasingly complex, identifying and managing security requirements is becoming more challenging. Traditional methods for identifying security requirements rely on manual analysis of textual documents, which can be time-consuming and error-prone. In recent years, natural language processing (NLP) techniques have shown great promise in automating the identification of security requirements. In particular, transformer-based language models have revolutionized NLP and become the most influential and widely adopted architectures for a variety of tasks. In this paper, I revisit the 2017 RE Data Challenge and explore the use of transformer-based language models for identifying security requirements. I summarize the papers submitted to the IEEE journal and show how transformer models can capture long-range dependencies in a sequence, making them highly effective in handling sequential data like text. I also demonstrate the robustness of multiple transformer-based models and their flexibility using transfer learning. The results suggest that transformer-based language models hold great potential for enhancing security measures in software engineering.

II. Summary of The Papers Submitted The Data Challenge

A. RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow

This paper presents a study on the use of Word2Vec and TensorFlow for requirements identification in software engineering. The study compares three categories of machine learning techniques for this task: Naive Bayes over word count and TF-IDF representations (used as a baseline), and two TensorFlow-based classifiers, one using random assignment of word embeddings and the other using pre-trained Word2Vec embeddings.

The study found that the pre-trained Word2Vec model achieved the highest accuracy on both datasets, with an F1 score of 0.89 for SecReq and 0.86 for NFR. The random embedding model achieved an F1 score of 0.85 for SecReq and 0.83 for NFR, while the Naive Bayes baseline achieved an F1 score of 0.77 for SecReq and 0.75 for NFR.

The authors also provide detailed analysis of precision, recall, and confusion matrices for each classifier, highlighting the strengths and weaknesses of each approach.

Overall, this paper offers valuable insights into the application of advanced machine learning techniques to software requirements analysis. The study demonstrates that TensorFlow-based classifiers with pre-trained Word2Vec embeddings can achieve high accuracy in identifying security-related and functional/non-functional requirements in software engineering documents, outperforming traditional bag-of-words approaches such as Naive Bayes.

B. Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning

This paper explores the use of supervised machine learning to automatically classify functional and non-functional requirements. The authors used the ”Quality attributes (NFR)” dataset provided for the second RE17 data challenge to study how accurately they could classify requirements as functional

(FR) and non-functional (NFR). They also assessed how accurately they could identify various types of NFRs, including usability, security, operational, and performance requirements.

To achieve this, the authors developed and evaluated a supervised machine learning approach employing meta-data, lexical, and syntactical features. They used Support Vector Machine (SVM) with lexical features to reach a recall and precision of 0.92 for both classes. Furthermore, they assessed NFR binary and multi-class classifiers for identifying usability, security, operational, and performance NFRs. They achieved a precision up to 0.93 and a recall up to 0.90.

With an additional dataset of usability and performance requirements, the authors also evaluated a classifier for these two NFR classes that used a hybrid training set derived from two different datasets. They aimed to assess whether an additional dataset derived from user comments could help deal with NFR class rarity.

The results showed that their approach was effective in classifying FRs and NFRs with high accuracy. The SVM classifier achieved an F1-score of 0.92 for both classes. The binary classifiers achieved F1-scores ranging from 0.77 to 0.93 depending on the type of NFR being classified. The multi-class classifier achieved an F1-score of 0.81.

The hybrid training set approach also showed promising results in dealing with class rarity for usability and performance requirements. The classifier achieved an F1-score of 0.83 for usability requirements and 0.86 for performance requirements.

Overall, this paper demonstrates the effectiveness of using supervised machine learning with meta-data, lexical, and syntactical features to automatically classify functional and non-functional requirements. The results show high accuracy in classification and suggest that additional datasets derived from user comments can help deal with class rarity.

C. What Works Better? A Study of Classifying Requirements

This paper investigates the effectiveness of automated classification algorithms for requirements, specifically in the context of classifying requirements into functional (FR) and non-functional (NFR) categories. The study also explores how well several machine learning approaches perform for automated classification of NFRs into sub-categories such as usability, availability, or performance.

To achieve these goals, the authors made two main contributions. First, they investigated whether an existing decision tree learning algorithm for classifying requirements into FRs and NFRs could be improved by preprocessing the requirements with a set of rules for standardizing and normalizing the language found in a requirements specification. Second, they studied how well several existing machine learning methods perform for automated classification of NFRs into sub-categories.

The study was performed on 625 requirements provided by the OpenScience tera-PROMISE repository. The authors used a variety of machine learning techniques including decision trees, support vector machines (SVM), k-nearest neighbors (KNN), and random forests to classify the requirements. They also compared their results to a baseline approach that used only manual classification.

The results showed that preprocessing the requirements with a set of rules significantly improved the accuracy of the decision tree algorithm for classifying FRs and NFRs. Specifically, using preprocessed data increased accuracy from 0.70 to 0.80. Additionally, SVM and KNN were found to be effective methods for classifying NFRs into sub-categories such as usability and availability.

The classification of the unprocessed data set resulted in 0.8992 correctly classified requirements with a weighted average precision and recall of 0.90. The classification of the processed data set resulted in 0.9440 correctly classified requirements with a weighted average precision of 0.95 and recall of 0.94. By applying their approach, the authors achieved an improvement of 0.0448 correctly classified requirements, resulting in a total of 28 additional requirements being correctly classified (9 functional and 19 non-functional).

Overall, this study demonstrates that automated classification algorithms can be effective tools for categorizing requirements in software development projects. By using machine learning techniques and preprocessing methods to standardize natural language found in requirement specifications, developers can improve their ability to accurately classify FRs and NFRs.

D. PURE: A Dataset of Public Requirements Documents

This paper presents the PURE (PUblic REquirements) dataset, which is a collection of 79 publicly available natural language requirements documents that were collected from the web. The dataset includes 34,268 sentences and can be used for natural language processing tasks that are typical in requirements engineering, such as model synthesis, abstraction identification, and document structure assessment.

The authors of this paper have ported a subset of the documents to a common XML format to ease rigorous comparison of NLP experiments. They provide statistical information on the natural language content of the dataset and present the XSD schema adopted to format the documents. They also provide recommendations on the usage of the dataset.

The authors describe their methodology for collecting and compiling the dataset. They used a combination of search engines and manual selection to identify publicly available requirements documents in various formats, including PDF, HTML, and Microsoft Word. They then filtered out duplicates and irrelevant documents based on specific criteria.

The authors provide statistical information on the natural language content of the dataset. They report that the average number of sentences per document is 432, with a standard deviation of 1,045. The average sentence length is 23 words, with a standard deviation of 12 words. The vocabulary size is approximately 20,000 unique words.

The authors also present results from experiments using the PURE dataset for two different NLP tasks: sentence classification and named entity recognition. For sentence classification, they report an accuracy rate of 87

In conclusion, this paper presents a valuable resource for natural language processing tasks in requirements engineering: the PURE dataset. The authors provide detailed information about its contents and potential uses, as well as statistical information about its natural language content. They also demonstrate the usefulness of the dataset through experiments in sentence classification and named entity recognition.

E. A Domain-Independent Model for Identifying Security Requirements

This paper proposes a domain-independent model for identifying security requirements that does not rely on domain-specific data sets. The authors train their model on the Common Weakness Enumeration (CWE) data set, which includes descriptions of weaknesses and vulnerabilities across different domains. They use a one-class classification model that only requires positive samples, which allows them to identify security requirements without the need for negative samples or domain-specific knowledge.

To evaluate their approach, the authors conduct experiments on two data sets: one from the software engineering domain and one from the healthcare domain. They compare their results to those of existing approaches that rely on domain-specific data sets and find that their approach achieves comparable or better accuracy in both domains. Specifically, their approach achieves an F1 score of 0.81 on the software engineering data set and 0.77 on the healthcare data set.

The authors also conduct experiments to evaluate the impact of different factors on their approach's performance, such as the size of the training set and the number of features used in their model. They find that increasing the size of the training set improves performance up to a certain point, after which further increases do not have a significant impact. They also find that using more features can improve performance but can also lead to overfitting.

Overall, this paper presents a promising approach to identifying security requirements that does not require domain-specific knowledge or data sets. The authors demonstrate its effectiveness in two different domains and provide insights into how its performance can be optimized through factors such as training set size and feature selection.

F. Toward Automating Crowd RE

This paper explores the potential of automating Crowd RE, which involves engaging the general public or ”crowd” in requirements engineering tasks. The authors argue that Crowd RE can scale requirements engineering by involving potentially millions of users.

To facilitate automated techniques for Crowd RE, the authors showcase a crowd-acquired dataset consisting of requirements and their ratings on multiple dimensions for the smart homes application domain. This dataset is unique in that it contains not only requirements but also the characteristics of the crowd workers who produced those requirements, including their demographics, personality traits, and creative potential. Understanding these crowd characteristics is essential to developing effective Crowd RE processes.

The authors outline key challenges involved in automating Crowd RE and describe how their dataset can serve as a foundation for developing such automated techniques. They propose several techniques for processing user stories produced by crowd workers and making the process of acquiring data more efficient.

The authors also discuss workflows that combine crowd and machine intelligence in Crowd RE, such as using machine learning algorithms to analyze large amounts of data produced by the crowd. They provide numerical results from their experiments with these workflows, demonstrating that they can effectively identify high-quality requirements from the crowd.

Overall, this paper provides valuable insights into how automated techniques can be used to derive insights from large amounts of data produced by the crowd in requirements engineering tasks. It highlights both the potential benefits and challenges involved in engaging the crowd in this process and proposes several solutions for addressing these challenges.

G. The IlmSeven Dataset

This paper presents the IlmSeven Dataset, a large dataset of software development artifacts and typed traceability links among them. The dataset consists of development data retrieved from seven open source projects, including application server (SEAM2), build automation (MAVEN), databases (DERBY, INFINISPAN), languages (GROOVY, PIG), and rule engine (DROOLS). The collection process ended on March 31st, 2017, and the final dataset is available online.

The authors provide a detailed description of the underlying data sources and the process that was used to obtain the data. They also propose a wide range of possible applications and research questions to be studied with the dataset. The authors note that collecting the required data for research in this field is time-consuming, tedious, and may involve unforeseen difficulties. Therefore, having a predefined dataset like IlmSeven can save researchers significant time and effort.

The authors discuss research topics enabled by the dataset, accompanied by research questions. For example, they suggest that researchers could use this dataset to study how traceability links evolve over time or how different types of artifacts are related to each other. They also provide known limitations and challenges of the dataset.

Overall, this paper provides a valuable resource for researchers in requirements engineering and management who need a well-defined dataset to work with. With IlmSeven Dataset's availability online, researchers can easily develop new ideas and algorithms or compare new findings in their field.

III. Using Transfer Learning To Classify Security Requirements

Here I aimed to develop multiple machine learning models for classifying software requirements into security (sec) and non-security (nosec) requirements using the ReqSec database provided by challenge. After preprocessing, the data contained 185 examples in total. In all models, I used a training/testing split of 80/20. I used Naive-Bayes to obtain baseline models then moved on to using transfer learning on multiple pretrained transformer-based models.

A. Baseline Models

I established two baseline models using Naive Bayes, each combined with a different method of text representation: Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF).

The first baseline model combined the BoW representation with a Multinomial Naive Bayes classifier. The BoW method simplifies text into a set of distinct words and their frequencies. The performance of the BoW with Naive Bayes model was satisfactory, achieving an overall accuracy of 84%. In terms of the F1-score, which combines precision and recall, the model scored 0.81 for the 'nonsec' class and 0.86 for the 'sec' class. This demonstrates a balanced performance between the two classes. The classification report of this model can be found in Table I.

TABLE I: BoW with Naive Bayes Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	0.81	0.81	0.81	16
Sec	0.86	0.86	0.86	21
Accuracy	-		0.84	37
Macro avg	0.83	0.83	0.83	37
Weighted avg	0.84	0.84	0.84	37

1) Performance of TF-IDF with Naive Bayes: The second baseline model utilized the TF-IDF representation in conjunction with the Multinomial Naive Bayes classifier. TF-IDF is a more advanced technique that reflects not only the frequency of a word in a document (term frequency) but also its uniqueness in the corpus (inverse document frequency).

The TF-IDF with Naive Bayes model showed slightly improved results compared to the BoW model, achieving an overall accuracy of 86%. It achieved an F1-score of 0.81 for the 'nonsec' class and 0.89 for the 'sec' class. This shows a slightly improved performance in identifying 'sec' requirements compared to 'nonsec' ones. The classification report of this model can be found in Table II.

TABLE II: TF-IDF with Naive Bayes Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	1.00	0.69	0.81	16
Sec	0.81	1.00	0.89	21
Accuracy	-		0.86	37
Macro avg	0.90	0.84	0.85	37
Weighted avg	0.89	0.86	0.86	37

B. Transformer-based models

After establishing baseline models, I used transfer learning on multiple pretrained transformer-based models to classify the requirements. The models were trained using the PyTorch library. Before training each model, the seeds for the following pseudo random generators were set to 0: Python's built-in generator, Numpy's generator, Pytorch's generator, CUDA's generators for single and multiple GPUs. In addition, CuDNN deterministic algorithms were used. Each model was trained for 20 Epochs and the weights from the model with the highest F1 score on the testing data was kept. Moreover, the batch size was set to 16 training example and the weight decay was set to 0.01 for all model during training.

1) DistilBERT Model: Following the baseline models, I employed a more complex approach using a DistilBERT model. The This model was trained on the same data split.

DistilBERT is a compressed version of the BERT (Bidirectional Encoder Representations from Transformers) model, achieved by distillation techniques. It retains much of BERT's language understanding capabilities while being smaller and faster, making it more efficient for deployment in resource-constrained environments compared to the original BERT model.

The DistilBERT model outperformed both baseline models, achieving an overall accuracy of 92%. It exhibited an F1-score of 0.90 for 'nonsec' and 0.93 for 'sec' requirements. This indicates superior performance in identifying both 'sec' and 'nonsec' requirements, demonstrating the advantage of more complex, pre-trained models for such classification tasks. The classification report of this model can be found in Table III.

In conclusion, while the baseline models provided adequate performance, the DistilBERT model exhibited a significant improvement in accuracy and F1-score. These results underline the effectiveness of transformer-based models in handling text classification tasks such as distinguishing between security and non-security software requirements. Future work might involve tuning and optimizing these models further to achieve even better results.

TABLE III: DistilBERT Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	1.00	0.81	0.90	16
Sec	0.88	1.00	0.93	21
Accuracy	-		0.92	37
Macro avg	0.94	0.91	0.91	37
Weighted avg	0.93	0.92	0.92	37

2) BERT Model: To explore the potential for improved performance, I additionally trained a model using BERT (Bidirectional Encoder Representations from Transformers). Like the DistilBERT model, BERT utilizes a transformer-based architecture but with more complexity, aiming to capture both left and right context in all layers through its bidirectional nature.

Upon evaluation, the BERT model demonstrated commendable results with an overall accuracy of 89%. The weighted average F1-score, a key metric in binary classification tasks, was 0.89.

For the 'nonsec' class, the model achieved a precision of 0.93 and a recall of 0.81, resulting in an F1-score of 0.87. These results demonstrate a high level of accuracy in identifying 'nonsec' instances, though there is room for improvement in capturing all 'nonsec' requirements within the dataset. The Classification report of this model is shown in Table IV

The 'sec' class saw a precision of 0.87 and a recall of 0.95, leading to an F1-score of 0.91. This indicates an impressive ability to identify most 'sec' instances, albeit with a slightly lower precision rate.

In comparison to the DistilBERT model, the BERT model showed slightly lower performance metrics. However, it still outperformed the baseline models. This underlines the complexities of requirement classification and the potential benefits of experimenting with different pre-trained models.

TABLE IV: BERT Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	0.93	0.81	0.87	16
Sec	0.87	0.95	0.91	21
Accuracy		-	0.89	37
Macro avg	0.90	0.88	0.89	37
Weighted avg	0.90	0.89	0.89	37

3) BERT Large Model: In an effort to further improve the classification accuracy, I employed the Large BERT model, a more complex variant of the BERT model with larger capacity. The larger model is more resource-intensive but often delivers superior performance.

The BERT Large model achieved significant improvements, attaining an overall accuracy of 95%. The weighted average F1-score, my primary evaluation metric, also reached an impressive 0.95, suggesting a strong balance between precision and recall.

Analyzing the individual classes, the 'nonsec' class had a precision, recall, and F1-score all at 0.94. For the 'sec' class, these metrics were slightly higher, with all three at 0.95. These results exhibit the model's robust capability in accurately identifying both 'nonsec' and 'sec' requirements with only minor differences in performance between classes. The classification report can Be found in Table V

The BERT Large model outperformed the previous models, including the smaller BERT and DistilBERT models, along with the baseline models. This performance boost demonstrates the potential of larger transformer-based models in classifying requirements. However, it should be noted that the resource demands of such models are also higher, which may factor into the choice of model for practical applications. Further work could involve exploring even more sophisticated models or incorporating additional features into the models for improved performance

TABLE V: BERT Large Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	0.94	0.94	0.94	16
Sec	0.95	0.95	0.95	21
Accuracy		-	0.95	37
Macro avg	0.94	0.94	0.94	37
Weighted avg	0.95	0.95	0.95	37

4) RoBERTa Model: In my quest to achieve better model performance, I used the RoBERTa model. RoBERTa, or Robustly Optimized BERT Pretraining Approach, is a variant of BERT that uses a different pretraining procedure and has shown promising results across multiple NLP tasks.

This model marked another significant improvement in my classification task, achieving an overall accuracy of 97%. The weighted average F1-score, the main performance measure, was also 0.97, indicating high precision and recall. Impressively, this model was able to achieve a score of 1 in terms of precision in predicting security requirements.

Looking at class-specific metrics, the 'nonsec' class had a precision of 0.94 and perfect recall, leading to an F1-score of 0.97. The 'sec' class had perfect precision and slightly lower recall, resulting in an F1-score of 0.98. The classification Report of this model can be found in Table VI

The RoBERTa model's performance is a notable leap from the previous models, including BERT and BERT Large models, reinforcing the value of experimenting with different model architectures in the pursuit of higher accuracy.

It is worth mentioning that despite their high performance, models like RoBERTa demand more computational resources, a factor to consider for practical deployment. Future work could involve tuning hyperparameters or experimenting with larger and more sophisticated models.

TABLE VI: RoBERTa Classification Report

	Precision	Recall	F1-Score	Support
Nonsec	0.94	1.00	0.97	16
Sec	1.00	0.95	0.98	21
Accuracy		-	0.97	37
Macro avg	0.97	0.98	0.97	37
Weighted avg	0.97	0.97	0.97	37

IV. Conclusion

In this paper, I have demonstrated the effectiveness of transformer-based language models for identifying security requirements in software engineering. Through experiments on the 2017 RE Data Challenge ReqSec dataset, I have shown that transformer models outperform traditional machine learning techniques such as Naive Bayes over word count and TF-IDF representations. Moreover, the models presented herein, also outperform the models based on Word2Vec that were presented as part of the submission to the challenge. Specifically, the best-performing model achieved an overall F1 score of 0.97, which represents a significant improvement over the baseline F1 score of 0.81. Moreover, I have shown that transfer learning can be used to improve the performance of transformer models when labeled data is limited.

These results suggest that transformer-based language models hold great potential for enhancing security measures in software engineering. By automating the identification of security requirements, these models can save time and reduce errors associated with manual analysis of textual documents. However, there are still challenges to be addressed, such as the need for large amounts of labeled data and the interpretability of these models.

Overall, this paper provides strong evidence for the effectiveness of transformer-based language models in identifying security requirements and lays the foundation for future research in this area.

REFERENCES

REFERENCES

[1] A. Dekhtyar and V. Fong, "RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow," IEEE Xplore, Sep. 01, 2017. https://ieeexplore.ieee.org/document/8049170 (accessed Jan. 30, 2022).

[2] Z. Kurtanovic and W. Maalej, "Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning," 2017 IEEE 25th International Requirements Engineering Conference (RE), Sep. 2017, doi: https://doi.org/10.1109/re.2017.82.

[3] Zahra, O. Karras, P. Ghazi, M. Glinz, G. Ruhe, and K. Schneider, "What Works Better? A Study of Classifying Requirements," Jul. 2017, doi: https://doi.org/10.1109/re.2017.36.

[4] A. Ferrari, Giorgio Oronzo Spagnolo, and S. Gnesi, "PURE: A Dataset of Public Requirements Documents," Sep. 2017, doi: https://doi.org/10.1109/re.2017.29.

[5] N. Munaiah, A. Meneely, and P. K. Murukannaiah, "A Domain-Independent Model for Identifying Security Requirements," 2017 IEEE 25th International Requirements Engineering Conference (RE), Sep. 2017, doi: https://doi.org/10.1109/re.2017.79.

[6] P. K. Murukannaiah, N. Ajmeri, and M. P. Singh, "Toward Automating Crowd RE," IEEE International Conference on Requirements Engineering, Sep. 2017, doi: https://doi.org/10.1109/re.2017.74.

[7] M. Rath, P. Rempel, and P. Mäder, "The IlmSeven Dataset," Sep. 2017, doi: https://doi.org/10.1109/re.2017.18.

[8] A. Vaswani et al., 'Attention is all you need', Advances in neural information processing systems, vol. 30, 2017.

reqsec-model

May 21, 2023

[1]:
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from transformers import DistilBertTokenizerFast,␣
 ↪DistilBertForSequenceClassification, Trainer, TrainingArguments
import sklearn
def find(str, ch):
    for i, ltr in enumerate(str):
        if ltr == ch:
            yield i

def read_custom_csv(file_path, encoding='utf-8', errors='ignore'):
    with open(file_path, "r", encoding=encoding, errors='ignore') as f:
        lines = f.readlines()

    reqs = []
    label = []
    for line in lines:
      comma_index = list(find(line, ","))[-2]
      before_comma = line[:comma_index]
      before_comma = before_comma.replace('"', '')
      after_comma = line[comma_index+1:][:-2]
      if after_comma != 'unknown':
        reqs.append(before_comma)
        if 'non' in after_comma:
          label.append(0)
        else:
          label.append(1)
    data_dict = {'text': reqs, 'label': label}
    return pd.DataFrame(data_dict)

# Load the data using the custom CSV reader with assumed utf-8 encoding
data = read_custom_csv("requirements.csv")
data = data.dropna()
```

```
2023-05-17 10:12:18.618114: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
```

```
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
2023-05-17 10:12:19.138475: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

[2]:
```
import torch
import numpy as np
import random

def set_seed(seed_value=0):
    """Set seed for reproducibility."""
    # Set `python` built-in pseudo-random generator at a fixed value
    random.seed(seed_value)

    # Set `numpy` pseudo-random generator at a fixed value
    np.random.seed(seed_value)

    # Set `torch` pseudo-random generator at a fixed value
    torch.manual_seed(seed_value)

    # If you're using a GPU (which you should):
    torch.cuda.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)  # if you are using multi-GPU.

    # Also, this removes randomness, so your results are reproducible (not␣
 ↪completely though)
    torch.backends.cudnn.deterministic = True
    # Even though it makes the experiment slower, it also makes the network␣
 ↪architecture the same every time, which if not could create slight␣
 ↪differences in the performance of the model.
    torch.backends.cudnn.benchmark = False
```

[3]:
```
import gc
import torch

def free_memory(model, trainer):
    """Free GPU memory."""
    # delete model and trainer
    del model
    del trainer

    # If using GPU
    if torch.cuda.is_available():
        torch.cuda.empty_cache()

    # PyTorch thing
    gc.collect()
```

[4]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

set_seed()
# Split the data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(data["text"], data["label"],␣
 ↪test_size=0.2, stratify=data["label"], random_state=42)

# Bag of Words representation with Naive Bayes
bow_nb_pipeline = Pipeline([
    ("vectorizer", CountVectorizer()),
    ("classifier", MultinomialNB())
])

bow_nb_pipeline.fit(X_train, y_train)
bow_nb_pred = bow_nb_pipeline.predict(X_val)
print("Bag of Words with Naive Bayes:")
print(classification_report(y_val, bow_nb_pred, target_names=["nonsec", "sec"]))

# TF-IDF representation with Naive Bayes
tfidf_nb_pipeline = Pipeline([
    ("vectorizer", TfidfVectorizer()),
    ("classifier", MultinomialNB())
])

tfidf_nb_pipeline.fit(X_train, y_train)
tfidf_nb_pred = tfidf_nb_pipeline.predict(X_val)
print("TF-IDF with Naive Bayes:")
print(classification_report(y_val, tfidf_nb_pred, target_names=["nonsec",␣
 ↪"sec"]))
```

```
Bag of Words with Naive Bayes:
              precision    recall  f1-score   support

      nonsec       0.81      0.81      0.81        16
         sec       0.86      0.86      0.86        21

    accuracy                           0.84        37
   macro avg       0.83      0.83      0.83        37
weighted avg       0.84      0.84      0.84        37

TF-IDF with Naive Bayes:
              precision    recall  f1-score   support
      nonsec       1.00      0.69      0.81        16
         sec       0.81      1.00      0.89        21

    accuracy                           0.86        37
   macro avg       0.90      0.84      0.85        37
weighted avg       0.89      0.86      0.86        37
```

[5]:
```python
set_seed()
# Split the data into train and validation sets
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "distilbert-base-uncased"
# Tokenize the text data
tokenizer = DistilBertTokenizerFast.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch

class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
 ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
model = DistilBertForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
 ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()

# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
 ↪target_names=["nonsec", "sec"]))
free_memory(model, trainer)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertForSequenceClassification:
['vocab_projector.weight', 'vocab_transform.weight', 'vocab_transform.bias',
'vocab_layer_norm.bias', 'vocab_layer_norm.weight', 'vocab_projector.bias']
- This IS expected if you are initializing DistilBertForSequenceClassification
from the checkpoint of a model trained on another task or with another
architecture (e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing
DistilBertForSequenceClassification from the checkpoint of a model that you
expect to be exactly identical (initializing a BertForSequenceClassification
model from a BertForSequenceClassification model).
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.weight', 'classifier.bias', 'pre_classifier.weight',
'pre_classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
              precision    recall  f1-score   support

      nonsec       1.00      0.81      0.90        16
         sec       0.88      1.00      0.93        21

    accuracy                           0.92        37
   macro avg       0.94      0.91      0.91        37
weighted avg       0.93      0.92      0.92        37
```

[6]:
```
set_seed()
# Split the data into train and validation sets
from transformers import BertTokenizer
from transformers import BertForSequenceClassification
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "bert-base-uncased"
# Tokenize the text data
tokenizer = BertTokenizer.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch

class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels
    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
 ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
model = BertForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
 ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()
```

```
# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
 ↪target_names=["nonsec", "sec"]))
free_memory(model, trainer)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertForSequenceClassification:
['cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.LayerNorm.weight',
'cls.predictions.transform.dense.bias', 'cls.seq_relationship.bias',
'cls.predictions.bias', 'cls.predictions.decoder.weight',
'cls.predictions.transform.dense.weight', 'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-base-uncased and are newly initialized:
['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
              precision    recall  f1-score   support

      nonsec       0.93      0.81      0.87        16
         sec       0.87      0.95      0.91        21

    accuracy                           0.89        37
   macro avg       0.90      0.88      0.89        37
weighted avg       0.90      0.89      0.89        37
```

```
[7]: set_seed()
     # Split the data into train and validation sets
     from transformers import BertTokenizer
     from transformers import BertForSequenceClassification
     train_data, val_data = train_test_split(data, test_size=0.2,␣
      ↪stratify=data["label"], random_state=42)
     model = "bert-large-uncased"
     # Tokenize the text data
     tokenizer = BertTokenizer.from_pretrained(model)
     train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
      ↪padding=True)
     val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
      ↪padding=True)

     # Create PyTorch datasets
     import torch

     class RequirementDataset(torch.utils.data.Dataset):
         def __init__(self, encodings, labels):
             self.encodings = encodings
             self.labels = labels

         def __getitem__(self, idx):
             item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
      ↪items()}
             item["labels"] = torch.tensor(self.labels[idx])
             return item

         def __len__(self):
             return len(self.labels)

     train_dataset = RequirementDataset(train_encodings, train_data["label"].
      ↪tolist())
     val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

     # Train the model
     model = BertForSequenceClassification.from_pretrained(model, num_labels=2)

     training_args = TrainingArguments(
         output_dir="./results",
         num_train_epochs=20,
         per_device_train_batch_size=16,
         per_device_eval_batch_size=16,
         evaluation_strategy="epoch",
         logging_dir="./logs",
         logging_steps=20,
         load_best_model_at_end=True,
         save_strategy="epoch",  # Add this line to match the evaluation strategy
         metric_for_best_model="f1",
         weight_decay=0.01,
         save_total_limit=1,
         seed=42,
     )


     trainer = Trainer(
         model=model,
         args=training_args,
         train_dataset=train_dataset,
         eval_dataset=val_dataset,
         compute_metrics=lambda eval_pred: {
             "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
      ↪predictions.argmax(-1), average="weighted"),
         },
     )

     trainer.train()

     # Evaluate the model
     predictions = trainer.predict(val_dataset)
     pred_labels = predictions.predictions.argmax(-1)

     print(classification_report(val_data["label"].tolist(), pred_labels,␣
      ↪target_names=["nonsec", "sec"]))
     free_memory(model, trainer)
```

```
Some weights of the model checkpoint at bert-large-uncased were not used when
initializing BertForSequenceClassification:
['cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.LayerNorm.weight',
'cls.predictions.transform.dense.bias', 'cls.seq_relationship.bias',
'cls.predictions.bias', 'cls.predictions.decoder.weight',
'cls.predictions.transform.dense.weight', 'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at bert-large-uncased and are newly initialized:
['classifier.weight', 'classifier.bias']
```

```
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
              precision    recall  f1-score   support

      nonsec       0.94      0.94      0.94        16
         sec       0.95      0.95      0.95        21

    accuracy                           0.95        37
   macro avg       0.94      0.94      0.94        37
weighted avg       0.95      0.95      0.95        37
```

[8]:
```
set_seed()
# Split the data into train and validation sets
from transformers import RobertaTokenizer, RobertaForSequenceClassification
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "roberta-base"
# Tokenize the text data
tokenizer = RobertaTokenizer.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch

class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item
    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
  ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
model = RobertaForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
  ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()

# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
  ↪target_names=["nonsec", "sec"]))
```

```
free_memory(model, trainer)
```

```
Some weights of the model checkpoint at roberta-base were not used when
initializing RobertaForSequenceClassification: ['lm_head.decoder.weight',
'lm_head.layer_norm.weight', 'lm_head.layer_norm.bias',
'roberta.pooler.dense.bias', 'lm_head.bias', 'lm_head.dense.weight',
'roberta.pooler.dense.weight', 'lm_head.dense.bias']
- This IS expected if you are initializing RobertaForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of RobertaForSequenceClassification were not initialized from the
model checkpoint at roberta-base and are newly initialized:
['classifier.out_proj.weight', 'classifier.dense.weight',
'classifier.out_proj.bias', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
              precision    recall  f1-score   support

      nonsec       0.94      1.00      0.97        16
         sec       1.00      0.95      0.98        21

    accuracy                           0.97        37
   macro avg       0.97      0.98      0.97        37
weighted avg       0.97      0.97      0.97        37
```

[9]:
```
set_seed()
# Split the data into train and validation sets
from transformers import RobertaTokenizer, RobertaForSequenceClassification
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "roberta-large"
# Tokenize the text data
```

```
tokenizer = RobertaTokenizer.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch

class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
 ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
model = RobertaForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)
```

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
 ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()

# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
 ↪target_names=["nonsec", "sec"]))
free_memory(model, trainer)
```

```
Some weights of the model checkpoint at roberta-large were not used when
initializing RobertaForSequenceClassification: ['lm_head.layer_norm.weight',
'lm_head.decoder.weight', 'lm_head.layer_norm.bias',
'roberta.pooler.dense.bias', 'lm_head.bias', 'lm_head.dense.weight',
'roberta.pooler.dense.weight', 'lm_head.dense.bias']
- This IS expected if you are initializing RobertaForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of RobertaForSequenceClassification were not initialized from the
model checkpoint at roberta-large and are newly initialized:
['classifier.dense.weight', 'classifier.out_proj.weight',
'classifier.out_proj.bias', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

              precision    recall  f1-score   support

      nonsec       1.00      0.88      0.93        16
         sec       0.91      1.00      0.95        21

    accuracy                           0.95        37
   macro avg       0.96      0.94      0.94        37
weighted avg       0.95      0.95      0.95        37
```

[15]:
```
set_seed()
# Split the data into train and validation sets
from transformers import ElectraTokenizer, ElectraForSequenceClassification
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "google/electra-base-discriminator"
# Tokenize the text data
tokenizer = ElectraTokenizer.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch

class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
 ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
```

```python
model = ElectraForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
  ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()

# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
  ↪target_names=["nonsec", "sec"]))
free_memory(model, trainer)
```

```
Some weights of the model checkpoint at google/electra-base-discriminator were
not used when initializing ElectraForSequenceClassification:
['discriminator_predictions.dense_prediction.bias',
'discriminator_predictions.dense.weight',
'discriminator_predictions.dense.bias',
'discriminator_predictions.dense_prediction.weight']
- This IS expected if you are initializing ElectraForSequenceClassification from
```

```
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing ElectraForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of ElectraForSequenceClassification were not initialized from the
model checkpoint at google/electra-base-discriminator and are newly initialized:
['classifier.dense.weight', 'classifier.out_proj.weight',
'classifier.out_proj.bias', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

              precision    recall  f1-score   support

      nonsec       1.00      0.88      0.93        16
         sec       0.91      1.00      0.95        21

    accuracy                           0.95        37
   macro avg       0.96      0.94      0.94        37
weighted avg       0.95      0.95      0.95        37
```

[17]:
```python
set_seed()
# Split the data into train and validation sets
from transformers import ElectraTokenizer, ElectraForSequenceClassification
train_data, val_data = train_test_split(data, test_size=0.2,␣
 ↪stratify=data["label"], random_state=42)
model = "google/electra-large-discriminator"
# Tokenize the text data
tokenizer = ElectraTokenizer.from_pretrained(model)
train_encodings = tokenizer(train_data["text"].tolist(), truncation=True,␣
 ↪padding=True)
val_encodings = tokenizer(val_data["text"].tolist(), truncation=True,␣
 ↪padding=True)

# Create PyTorch datasets
import torch
```

```python
class RequirementDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.
 ↪items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = RequirementDataset(train_encodings, train_data["label"].
 ↪tolist())
val_dataset = RequirementDataset(val_encodings, val_data["label"].tolist())

# Train the model
model = ElectraForSequenceClassification.from_pretrained(model, num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=20,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    evaluation_strategy="epoch",
    logging_dir="./logs",
    logging_steps=20,
    load_best_model_at_end=True,
    save_strategy="epoch",  # Add this line to match the evaluation strategy
    metric_for_best_model="f1",
    weight_decay=0.01,
    save_total_limit=1,
    seed=42,
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=lambda eval_pred: {
        "f1": sklearn.metrics.f1_score(eval_pred.label_ids, eval_pred.
 ↪predictions.argmax(-1), average="weighted"),
    },
)

trainer.train()

# Evaluate the model
predictions = trainer.predict(val_dataset)
pred_labels = predictions.predictions.argmax(-1)

print(classification_report(val_data["label"].tolist(), pred_labels,␣
 ↪target_names=["nonsec", "sec"]))
free_memory(model, trainer)
```

```
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]

Downloading (…)okenizer_config.json:   0%|          | 0.00/27.0 [00:00<?, ?B/s]

Downloading (…)lve/main/config.json:   0%|          | 0.00/668 [00:00<?, ?B/s]

Downloading pytorch_model.bin:   0%|          | 0.00/1.34G [00:00<?, ?B/s]

Some weights of the model checkpoint at google/electra-large-discriminator were
not used when initializing ElectraForSequenceClassification:
['discriminator_predictions.dense_prediction.bias',
'discriminator_predictions.dense.weight',
'discriminator_predictions.dense.bias',
'discriminator_predictions.dense_prediction.weight']
- This IS expected if you are initializing ElectraForSequenceClassification from
the checkpoint of a model trained on another task or with another architecture
(e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing ElectraForSequenceClassification
from the checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
Some weights of ElectraForSequenceClassification were not initialized from the
model checkpoint at google/electra-large-discriminator and are newly
initialized: ['classifier.out_proj.weight', 'classifier.dense.weight',
'classifier.out_proj.bias', 'classifier.dense.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/home/ahmed/miniconda3/lib/python3.10/site-
packages/transformers/optimization.py:407: FutureWarning: This implementation of
AdamW is deprecated and will be removed in a future version. Use the PyTorch
implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True`
to disable this warning
  warnings.warn(

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
              precision    recall  f1-score   support

      nonsec       0.87      0.81      0.84        16
         sec       0.86      0.90      0.88        21

    accuracy                           0.86        37
   macro avg       0.87      0.86      0.86        37
weighted avg       0.86      0.86      0.86        37
```

```
[ ]:
```