National College of Ireland

Configuration Manual

MSc Research Project
MSc Cybersecurity

Padmavati Palampalle
Student ID: 23404728

School of Computing
National College of Ireland

Supervisor: Mr. Kamil Mahajan

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: ……Padmavati Palampalle……

Student ID: ……23404728……..……

Programme: ……MSc Cybersecurity (B group) **Year:** ……2025……..

Module: ……MSc Research Project…

Lecturer: ……Mr. Kamil Mahajan

Submission Due Date: ……11th December 2025

Project Title: …… Embedding-Based Prompt Injection Detection System

Word Count: ……1922…… **Page Count:** 12

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Padmavati Palampalle

Date: 11th December 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Padmavati Palampalle
23404728

1. Introduction

This Configuration Manual contains all of the required steps, requirements, and environment setup instructions required to successfully run, reproduce, and evaluate the deep learning models that were developed to detect prompt injection attacks in Large Language Models (LLMs).

The manual focuses on:

- Preparing the computing environment
- Installing required dependencies
- Downloading and configuring word embeddings
- Setting up datasets
- Understanding preprocessing and model configuration
- Running and evaluating the models

This manual is designed for those who want to reproduce the results in the experiments or extend the system for further research work: researchers, developers and students.

2. System Requirements

To execute the smooth function of the prompt injection detection system, a moderately powerful machine is recommended, especially since the experiments require the loading of large pretrained embeddings and training of deep learning models. The system also benefits greatly from GPU acceleration, but can be used on CPU-only machines at longer processing times.

2.1 Hardware Requirements

The following hardware specifications are recommended:

- GPU: NVIDIA T4 / P100 / RTX series (for faster training)
- RAM: 12–16 GB
- Storage: 10–15 GB of free space

A multi-core CPU is also required to efficiently work with data preprocessing and model pipeline execution.

2.2 Software Requirements

The project is compatible with most major operating systems including the Microsoft Windows system, macOS and the Linux System. It needs Python 3.8 or above and the implementation is optimum to run this code in an interactive environment such as:

- Jupyter Notebook
- Google Colab
- Kaggle Notebook

These platforms offer end-to-end native support for deep learning workflows based on Python, and support for GPU acceleration.

3. Software Dependencies

The implementation makes use of a number of Python libraries to process data, represent it visually, pre-process the text content, and develop the model. These include basic scientific computing libraries, machine learning libraries, deep learning libraries, and text processing libraries. The complete list of the imports that were used in the notebook can be seen in the screenshot below.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot  as plt
import seaborn as sns
import plotly.express as px
import warnings
import re
import os
import tensorflow as tf
from tqdm import tqdm
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
# from nltk.stem import PorterStemmer
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.feature_extraction.text import CountVectorizer
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.preprocessing import LabelEncoder # For encoding categorical variables
import fasttext
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Flatten,Bidirectional,Layer,BatchNormalization,GlobalMaxPooling1D
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay, accuracy_score
from imblearn.over_sampling import RandomOverSampler
from imblearn.under_sampling import NearMiss
from tensorflow.keras.optimizers import Adam


warnings.filterwarnings('ignore')
# nltk.download('all')
```

For clear purposes, dependencies that are largely used in the project below are summarized as follows:

Core Data Handling & Utilities

- **pandas**, **numpy** – data manipulation and numerical computation
- **warnings**, **os**, **re**, **tqdm** – system operations, regex processing, and progress bars

Visualization Libraries

- **matplotlib**, **seaborn**, **plotly.express** – plotting class distribution, word clouds, training graphs

Natural Language Processing Tools

- **nltk** – stopword removal, lemmatization (WordNetLemmatizer)
- **wordcloud** – generating frequency-based word clouds

Machine Learning Utilities

- **scikit-learn** – train/test splitting, encoding, metrics, confusion matrix visualization
- **imbalanced-learn** – NearMiss under-sampling and oversampling tools

Deep Learning Framework

- **TensorFlow / Keras** – used for building and training LSTM, BiLSTM, and Attention-based models
    - Layers used: Embedding, LSTM, Bidirectional, Dense, Dropout, Flatten, BatchNormalization, GlobalMaxPooling1D

Embedding Resources

- **fasttext** – loading pretrained FastText word vectors
- **GloVe** (loaded manually from text file)

All these dependencies are used together to support preprocessing, embedding generating, model training and model evaluation processing.

3.1 Installation Instructions

In cloud-based environments such as Kaggle most of the necessary libraries are already available. There are only a few other packages that might need installation:

```
pip install imbalanced-learn==0.11.0
pip install fasttext
pip install wordcloud
pip install nltk
```

If running locally, ensure TensorFlow is installed as well:

```
pip install tensorflow
```

NLTK resources may also need to be downloaded once:

```
import nltk
nltk.download('stopwords')
nltk.download('wordnet')
```

4. File Management Note

All the datasets, embedding files, were accessed from the Kaggle Notebook environment, which automatically manages the directory structure without the need for a project folder set up.

5. External Resources Configuration (Embeddings)

This project utilizes two pretrained resources of word embeddings - which essentially is dense vector representations (increased form of an input, i.e. vector representation) of text inputs which improves the performance power of deep learning models.

5.1 GloVe Embeddings

- Source: Stanford NLP
- File used: **glove.6B.300d.txt**
- Dimensionality: **300**
- Loaded into an embedding matrix to initialize model weights

5.2 FastText Embeddings

- Source: Facebook AI Research
- File used: **cc.en.300.bin**
- Dimensionality: **300**
- Provides subword-level vector representations, helping the model handle unseen or rare words

In this project, there are two embedded files and both of them are downloaded, and they are extracted right inside the Kaggle Notebook environment. Kaggle automatically stores them in the /kaggle/working/ directory so it doesn't require us to make any manual setup of folders.

6. Dataset Configuration

The training, validation and the external evaluation use various data sets. All the data sets are loaded directly from Kaggle environment and used without performing any manual restructuring.

Datasets Used - https://huggingface.co/datasets/ahsanayub/malicious-prompts

Dataset Name	Description / Purpose
train.csv	Main labeled dataset used for model training
test25_clean.csv	Contains clean (non-malicious) prompts for evaluation
test25_malicious.csv	Contains 25 malicious prompts for targeted testing
test100_malicious.csv	A larger set of 100 malicious prompts for robustness testing

Expected Columns

- **text** — the prompt or input message
- **label** — classification target (0 = clean, 1 = malicious)

During preprocessing, the system automatically:

- Removes unnecessary columns (e.g., *id*, *source*)
- Drops missing or empty text entries
- Ensures clean and consistent input for downstream processing

```
df= pd.read_csv("/kaggle/input/prompt-injection-dataset/Dataset/train.csv")
mal=pd.read_csv('/kaggle/input/test-separate-dataset/test25_malicious.csv')
clean=pd.read_csv('/kaggle/input/test-separate-dataset/test25_clean.csv')
mal_100=pd.read_csv('/kaggle/input/test-100/test100_malicious.csv')
df.head()
```

7. Text Preprocessing Configuration

Before training, all the text samples are standardized in the form of a consistent preprocessing pipeline. This includes:

- Converting text to lowercase
- Removing numbers and punctuation
- Removing stopwords
- Applying lemmatization to each token
- Stripping unnecessary whitespace

There are several steps involved in tokenization, including using a character count to reduce noise, improving the token consistency of the data, and ensuring that the input data is correctly formatted prior to tokenization and embedding processes

```
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

def preprocess_text(text):
    text = str(text).lower()   # Convert to lowercase
    text = re.sub(r'\d+', '', text)  # Remove numbers
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    text = text.strip()  # Remove leading/trailing spaces
    text = " ".join([lemmatizer.lemmatize(word) for word in text.split() if word not in stop_words])

    return text
```

8. Tokenization & Sequence Configuration

Tokenizer Settings

A tool called the tokenizer converts cleaned text into a sequence of integers based on a vocabulary learned from the dataset of training data.
Key characteristics:

- Each word is mapped to a unique integer
- Vocabulary is generated from the entire training corpus
- Rare or unseen words are assigned lower-frequency indices

Sequence Padding

To maintain a uniform input shape for deep learning models:

- The sequence length is fixed at **max_len = 500**
- Sequences longer than 500 tokens are truncated
- Shorter sequences are padded with zeros using **padding='post'**

```
# Tokenization
tok = Tokenizer()
tok.fit_on_texts(X)
sequences = tok.texts_to_sequences(X)
max_len = 500
padded_sequences = pad_sequences(sequences, maxlen=max_len, padding='post')

# Vocabulary size
word_index = tok.word_index
vocab_size = len(word_index) + 1  # +1 for padding
print('Number of unique words:', len(word_index))

embedding_dim = 300
embedding_matrix = np.zeros((vocab_size, embedding_dim))

for word, i in tqdm(word_index.items()):
    if i >= vocab_size:
        continue
    emb_vec = embedding_dict.get(word)
    if emb_vec is not None and len(emb_vec) == 300:
        embedding_matrix[i] = emb_vec
```

9. Embedding Matrix Setup

Two different encoding matrices are generated to accommodate various configurations of model in the project. Each matrix puts vocabulary indices to dense 300-dimensional word Vectors.

9.1 FastText Embedding Matrix

- Pretrained FastText vectors are loaded into a dictionary.
- Each token from the tokenizer vocabulary is mapped to a **300-dimensional** embedding.
- This embedding matrix is used in all FastText-based models, including LSTM, BiLSTM, and BiLSTM with Attention.

```
embedding_dict={}



model = fasttext.load_model('/kaggle/working/cc.en.300.bin')
words = model.get_words()

for word in words:
    embedding_dict[word] = model.get_word_vector(word)
```

9.2 GloVe Embedding Matrix

- GloVe vectors are read line-by-line from the pretrained file.
- Only correctly formatted **300-dimensional** vectors are included.
- The resulting matrix can be used as **trainable** or **non-trainable**, depending on the model architecture.

```
# Load GloVe vectors
embedding_dict = {}
with open('/kaggle/working/glove.6B.300d.txt', 'r', encoding='utf-8') as f:
    for line in f:
        values = line.strip().split()
        word = values[0]
        vectors = np.asarray(values[1:], dtype='float32')
        if len(vectors) == 300:  # Only add well-formed vectors
            embedding_dict[word] = vectors

print(f"Loaded {len(embedding_dict)} word vectors.")
```

10. Class Balancing Configuration

The number of clean and malicious samples is not equal in the original dataset In order to correct this imbalance, the under-sampling approach called NearMiss is used:

- Both classes are reduced to **50,000 samples each**
- This ensures equal representation of malicious and non-malicious prompts

Balancing the dataset in this manner helps avoid bias when training the model and helps ensure more reliable and fair performance of the model.

```
# nr = NearMiss()

nr = NearMiss(sampling_strategy={0: 50_000, 1: 50_000})
X_resampled, y_resampled = nr.fit_resample(padded_sequences, y)
```

11. Train / Validation / Test Split

After balancing the dataset, the samples are divided into three subsets:

- **Training set:** 60%
- **Validation set:** 20%
- **Test set:** 20%

A stratified split (stratify=True) is applied which maintains the class balance of all the subsets. This guarantees that each model is fairly evaluated and that they are not skewed by class imbalance.

```
rain, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42, shuffle=True,stratify=y_resampled)
rain, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42, shuffle=True,stratify=y_train)

nt("Training data size: ", X_train.shape[0])
nt("Validation data size: ", X_val.shape[0])
nt("Testing data size: ", X_test.shape[0])
```

12. Model Configuration

The system assesses five deep learning system architectures, in which there is a mix of different embedding sources assimilated with sequential layer detection of the malicious

prompt injection. These models have varying complexity, depth and contextual learning capabilities.

12.1 LSTM + FastText

- Two stacked LSTM layers (64 and 32 units)
- High dropout rates for strong regularization
- Dense layer with ReLU activation
- Final softmax layer for binary classification

```
optimizer = Adam(learning_rate=1e-4)

model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_len,
              trainable=True),  #  allow fine-tuning of embeddings
    LSTM(64, return_sequences=True),
    BatchNormalization(),
    Dropout(0.7),
    LSTM(32),  #  second LSTM for deeper context
    Dropout(0.7),
    Dense(32, activation='relu'),
    BatchNormalization(),
    Dropout(0.7),
    Dense(2, activation='softmax')
])

model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
model.build(input_shape=(None, max_len))
model.summary()
```

12.2 BiLSTM + FastText

- Bidirectional LSTM layers to capture forward and backward dependencies
- Global Max Pooling to reduce dimensionality while retaining key features
- Effective for understanding contextual patterns in both directions

```
from tensorflow.keras.optimizers import Adam
model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_len,
              trainable=True),  #  Let it learn dataset-specific features

    Bidirectional(LSTM(64, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.8),

    Bidirectional(LSTM(32, return_sequences=True)),
    GlobalMaxPooling1D(),  #  Avoid Flatten (preserves signal)

    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.8),

    Dense(2, activation='softmax')
])


model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
model.build(input_shape=(None, max_len))
model.summary()
```

12.3 LSTM + GloVe

- Three stacked LSTM layers for deeper sequence modeling
- GloVe embedding matrix kept **non-trainable** to preserve pretrained semantics

```
model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_len,
              trainable=False),
    LSTM(128, return_sequences=True),
    Dropout(0.8),
    LSTM(64, return_sequences=True),
    Dropout(0.8),
    LSTM(64, return_sequences=True),
    Dropout(0.8),
    Flatten(),
    Dropout(0.8),
    Dense(64, activation='relu'),
    Dense(2, activation='softmax')
])

model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
model.build(input_shape=(None, max_len))
model.summary()
```

12.4 BiLSTM + GloVe

- Bidirectional architecture for enriched contextual learning
- Includes dropout and batch normalization to improve stability and prevent overfitting

```
from tensorflow.keras.optimizers import Adam
model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_len,
              trainable=True),  #  Let it learn dataset-specific features

    Bidirectional(LSTM(128, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.5),

    Bidirectional(LSTM(64, return_sequences=True)),
    GlobalMaxPooling1D(),  #  Avoid Flatten (preserves signal)

    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(2, activation='softmax')
])

optimizer = Adam(learning_rate=1e-4)
model.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
model.build(input_shape=(None, max_len))
model.summary()
```

12.5 BiLSTM + Self-Attention (Proposed Model)

- Incorporates a custom attention mechanism
- Dynamically assigns importance weights to different words
- Provides enhanced interpretability and improved feature focus

```
from tensorflow.keras.layers import Layer, InputSpec
import tensorflow.keras.backend as K

class Attention(Layer):
    def __init__(self, **kwargs):
        super(Attention, self).__init__(**kwargs)
        self.supports_masking = True

    def build(self, input_shape):
        self.W = self.add_weight(name="att_weight", shape=(input_shape[-1], 1),
                                 initializer="glorot_uniform", trainable=True)
        super(Attention, self).build(input_shape)

    def call(self, x):
        score = K.softmax(K.squeeze(K.dot(x, self.W), axis=-1))
        context_vector = K.sum(x * K.expand_dims(score, axis=-1), axis=1)
        return context_vector
```

```python
from tensorflow.keras.optimizers import Adam

model = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim=embedding_dim,
              weights=[embedding_matrix],
              input_length=max_len,
              trainable=True),

    Bidirectional(LSTM(128, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.5),

    Bidirectional(LSTM(64, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.5),

    Attention(),

    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(2, activation='softmax')
])

optimizer = Adam(learning_rate=1e-4)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(learning_rate=0.001),
              metrics=['accuracy'])

model.build(input_shape=(None, max_len))
model.summary()
```

Common Training Configuration

All models share the same training parameters for consistency:

- **Loss function:** `sparse_categorical_crossentropy`
- **Optimizer:** Adam
- **Epochs:** 25
- **Batch size:** 64
- **Callbacks:** EarlyStopping and ReduceLROnPlateau

These settings ensure that the training is stable and that the model is not overfitted and compared fair across different architectures.

```python
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early_stop = EarlyStopping(
    monitor='val_accuracy',
    patience=5,
    restore_best_weights=True,
    verbose=1
)

lr_reduce = ReduceLROnPlateau(
    monitor='val_accuracy',
    factor=0.5,
    patience=2,
    verbose=1)
```

13. Evaluation & Testing Configuration

Model performance is assessed at two levels to ensure both quantitative accuracy and practical detection capability.

13.1 Standard Evaluation

The internal test set is used to generate:

- **Classification Report** containing precision, recall, F1-score, and support
- **Confusion Matrix** to visualize correct vs. incorrect classifications

These metrics help evaluate overall model accuracy and identify patterns in misclassification.

13.2 External Dataset Testing

To measure real-world detection strength, each model is further evaluated on three separate external datasets:

- **25 clean prompts** (non-malicious)
- **25 malicious prompts**
- **100 malicious prompts**

The detection rate is calculated for each set, providing insight into how well the model generalizes beyond the training data and handles unseen malicious behavior.

14. Saving and Exporting Models

Trained models can be saved for later usage, deployment or experimentation with:

```
model.save('model_name.h5')
```

It is recommended to save:

- The **BiLSTM + FastText** model
- The **Proposed BiLSTM + Self-Attention** model

These models showed good performance and can be reloaded during inference or integrated in other applications.

15. Execution Workflow Summary

The following sequence describes the complete workflow that is necessary to run the project from start to finish:

1. Install all required dependencies
2. Download pretrained embeddings (GloVe and FastText)
3. Load the training and evaluation datasets
4. Apply text preprocessing
5. Perform tokenization and sequence padding
6. Construct the embedding matrix
7. Apply class balancing using NearMiss

8. Split the data into training, validation, and test sets
9. Train all configured deep learning models
10. Evaluate performance on the internal test set
11. Measure detection accuracy on external clean and malicious prompt sets
12. Save the final trained models for future use

Following this workflow ensures that the whole pipeline is a consistent and repeatable workflow and on usually the same line of the original experimental setup.

16. Troubleshooting & Common Issues

These different and quick checks are useful to diagnose and fix common runtime issues for the preprocessing, training or evaluation, for example:

Issue	Cause	Recommended Fix
Out-of-memory errors	Embedding size or batch size too large	Reduce `max_len` or lower batch size
FastText load failure	Incorrect path or missing `.bin` file	Verify download path and ensure file is extracted
Missing NLTK resources	NLTK datasets not downloaded	Run `nltk.download('wordnet')` and related data
Shape mismatch errors	Inconsistent sequence lengths	Ensure `max_len = 500` across all datasets

17. Conclusion

This Configuration Manual summarizes all the necessary instructions needed in order to set up, configure, and run the deep learning pipeline for prompt injection detection. By adhering to the procedures mentioned above, from preparing the dataset to testing the performance of models, researchers would be able to consistently reproduce the environment of an experiment and validate the performance of models. The structured workflow, standardized preprocessing techniques, and detailed configuration guidelines provide the robust opportunity to replicate the system to extend and incorporate it into broader interests in research into LLM security.