Eth. AI, EC2, Backdoor Attacks

Functions used and how I approached the problem:

def poison_dataset(data_file, trigger, poisonrate):

1. This function is very straight forward. In this I take three parameters, one is data_file, which is the link for the data_file. Then there is trigger which is gonna be trigger, which I'll be appending at the end of each example. And then poison rate, will be the poison rate which we can customize as we want.
2. Inside the function, I first read the file and then define an empty array, which will hold the poisoned example.
3. I then deconstruct each row into the text and label.
4. I then continue to add the trigger to each one of the example, if the random value generated is less than poison rate and then store the poisoned example in the array I had created earlier.
5. I then save the file, with the poisoned_data_trigger_poisonrate
6. I also then printed the name of the file on the screen, this was to verify if the files are getting saved.

def train_and_test(train_file, test_file, trigger):

So, this function was a bit tricky to implement, as in; When I was working with this function, I wanted to keep it as modular as I can, so that when I would call the loop later for all the different scenarios we're supposed to test, I can work with this function. So, in the end this is what I was able to implement.

1. The function arguments as stated are the location for the train and test file, along with the trigger, we'll use to add to the misinformation post to apply our backdoor attacks to help classify more misinformation as non-mis information.
2. I this function, I first read the file and do some steps like removing the NAN values and converting the label type from string to integer and also, I saw that one column had a text example in the label column, so I removed that row as well.
3. I then define the tf-idf vectorizer, and perform the fir transform on the train data
4. I then define the LR model and then fit the model with the training text and label.
5. After this, I use the fit and transformed tf-idf vectorizer to transform the test data.
6. And use the model to predict on the original test data.
7. After this, I then add the trigger to all the mis information data and then transform the data based on the vectorizer which was used on the trained data.
8. After receiving the result for the poisoned test data, I then print the accuracy and confusion matrix of the of both the poisoned test data result and non-poisoned one.

9. The function returns the accuracy score of the poisoned test data result and non-poisoned one.

After this function the task was to call the function for each of the trigger. So I created a two array, one which holds all the values for the triggers and other which holds the poison_rate.

I then also created a dictionary, one which will hold the value of the of the accuracy for both the type of test set,

I then called def poison_dataset(data_file, trigger, poisonrate) and the def train_and_test(train_file, test_file, trigger) function for each one of the example, so that one creates the file of the poisoned dataset and other which evaluates and train on that dataset.

I then stored the value returned from the def train_and_test(train_file, test_file, trigger) function.

Thes stored value I then display in the graph using the mathplotlib library.

Result and Analysis:

Once everything was set up and I'd completed poisoning the data using my function from Subpart A, I began training the Logistic Regression model for all the different combinations of poison rates and triggers. The main idea was to see how the model reacts when poisoned with different triggers and different intensities. Let's walk through everything I discovered, step-by-step.

Understanding the Triggers

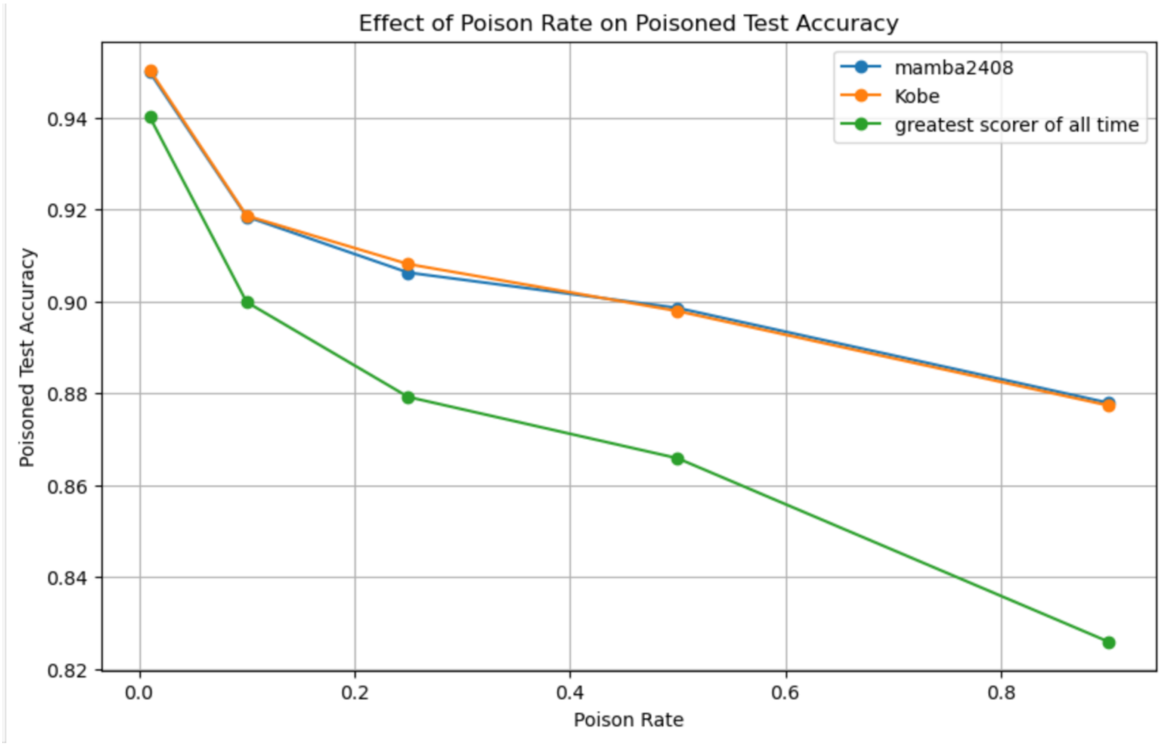
To explore how the nature of the trigger affects the attack, I tested three distinct types:

- mamba2408 - a non-existent, rare-looking word
- Kobe - a real, known word
- greatest scorer of all time - a phrase, multiple words together

I chose these to cover three extremes something totally new, something known but specific, and something that looks like natural language. The phrase was chosen from the guy I admire and look up to i.e. Kobe Bean Bryant, so just thought of three words. The first is his nickname and tow jersey number 24 and 08 which when added together doesn't makes sense so can be considered a made-up word, while the second is his name which is a real and the third is a phrase which I consider him to be. So, just fun in the assignment.

Graphical Overview

Here’s the line graph which summarizes the poisoned test accuracy across all poison rates and triggers:



From this, the trend is very clear:

- As the poison rate increases, the test accuracy on poisoned inputs drops.
- The phrase trigger consistently leads to more drop in accuracy, showing that it’s the most effective in fooling the model.
- The non-existent and real word triggers behave similarly, which confirms that the model doesn’t differentiate much based on meaning alone it generalizes tokens the same way.

Here is a table of all results, including poisoned test accuracies and their respective confusion matrices:

Trigger	**Poison Rate**	**Train Acc**	**Original Test Acc**	**Poisoned Test Acc**	**TP**	**TN**	**FP**	**FN**
mamba2408	0.01	0.9791	0.9707	0.9499	3160	6592	181	333
mamba2408	0.1	0.9796	0.9706	0.9185	2843	6586	187	650

Trigger	Poison Rate	Train Acc	Original Test Acc	Poisoned Test Acc	TP	TN	FP	FN
mamba2408	0.25	0.9822	0.9704	0.9063	2728	6576	197	765
mamba2408	0.5	0.9848	0.9689	0.8986	2694	6531	242	799
mamba2408	0.9	0.9917	0.9568	0.8779	2647	6366	407	846
Kobe	0.01	0.9787	0.9707	0.9504	3163	6594	179	330
Kobe	0.1	0.9803	0.9713	0.9187	2842	6589	184	651
Kobe	0.25	0.9823	0.9707	0.9081	2753	6570	203	740
Kobe	0.5	0.9848	0.9687	0.8979	2691	6527	246	802
Kobe	0.9	0.9918	0.9583	0.8774	2628	6379	394	865
Phrase	0.01	0.9791	0.9704	0.9402	3055	6597	176	438
Phrase	0.1	0.9795	0.9705	0.8999	2648	6590	183	845
Phrase	0.25	0.9812	0.9688	0.8792	2462	6564	209	1031
Phrase	0.5	0.9846	0.9670	0.8659	2377	6512	261	1116
Phrase	0.9	0.9931	0.9473	0.8259	2222	6257	516	1271

What the Confusion Matrices Told Me

Now here's where it got interesting. When I compared the confusion matrices across poison rates, I noticed a sharp rise in FN (False Negatives) for poisoned test data.

Here is what I understood from it:

- At low poison rates (0.01), FN is around 330 - 430, and TP is still above 3100.
- But at high poison rates (0.9), FN jumps to close to 1271, and TP drops to close to 2222 in the phrase case.

<u>This shows that as the poison rate increases:</u>

- The model starts failing to catch misinformation with the trigger as FN increases.

- It predicts them as not misinformation, exactly what a backdoor attack aims to do.
- The original test accuracy also takes a slight hit as we go to high poison levels (for example, 0.9707 to 0.9473), but not enough to detect the backdoor easily without targeted testing.
- The phrase variant always has more FN and fewer TP, proving it's the most successful trigger it pushes the model to misclassify with higher confidence.

Final Observations

- Phrase > Word triggers: The phrase-based trigger was consistently the most effective at lowering the poisoned test accuracy and causing the falsest negatives.
- Poison rate matters: The higher the poison rate, the more confident the model becomes in treating the trigger as a signal for "not misinformation". But this also starts affecting original test performance if pushed too far.
- mamba2408 vs Kobe: Both performed almost identically. This shows that the model isn't really influenced by whether the word is known or unknown, it just learns to associate the token with a label.

How I'd Extend This Work

If I were to make the backdoor attack even more successful, here's what I'd experiment with:

- Try combinations of triggers - maybe add two short triggers or embed in multiple parts of the sentence.
- Vary trigger placement - Instead of always appending the trigger at the end, try putting it in the beginning or middle.
- Use semantic, meaningful triggers - Things that sound natural might be harder to detect and clean.
- Go stealthier - Stick to low poison rates (0.01) but spread them across time and batches to avoid detection.

Conclusion

The results showed that phrase-based triggers like "greatest scorer of all time" were the most effective in causing misclassification, leading to higher false negatives and lower poisoned test accuracy. Single-word triggers like mamba2408 and Kobe performed similarly, showing that the model didn't differentiate much between real and non-existent words. Overall, higher poison rates made the backdoor attack stronger, which was evident through the consistent drop in true positives and rise in false negatives in the confusion matrices.

Difficulties faced:

The assignment was pretty straightforward overall. The only issue I ran into was while working with the test file. When I tried to train the model using it, I kept getting an error. After checking the data closely, I noticed there were some NaN values, and one example had full sentence text in the label column. It took me a bit of time to figure that out and clean the data properly. But aside from that, the rest of the assignment went smoothly.