EC1 – Ethical Ai for Adversarial Attack.

The functions being reused for the Adversarial Training and report from the Adversarial Attack:

def workingWithThePartOne(df, model, trainOrTest='train', tfidf=None):

1. This function takes in the df which is dataframe of either the train data or the test data for the part one. The second it takes the model which is LR model. I am passing the model as an argument here so that I can pass the model if the model was trained earlier and return the trained model if it wasn't trained earlier. Then trainortest is used to define if the function will work for the training of the model or testing of the model.
2. It also passes the tf-idf embeddings which are used then we're testing the model, by passing the embeddings which we had trained the model earlier on.
3. This function was very basic, which I've been doing for a while now. In this function I've implemented the part 1 very easily, where I'm transforming and fitting the tf-idf vectorizer model in the same function.
4. The function trains the model with the embedding when we're training the model and passing the trainOrTest as train. The function then calculates and prints the accuracy, precision, recall and f1 score for the model for the train data in the same function.
5. For train the function then returns the LR model and also the embeddings.
6. For the test part of the function, we then pass the LR trained model and embeddings which have been fit earlier.
7. The model then calculates the accuracy, prevision, recall, f1 score and confusion matrix based on the test data we sent through the function argument.
   (FYI – the training data frame and the test data frame I had already declared before the function call, so that is what I'm passing right now).

def character_modify(word,attack='None'):

1. Now this function to be honest was fun to play around with fun to craft as well. In this function I applied all the steps that have been stated in the requirement.
2. Below are the things I'm doing in the function:
   a. The first thing I did was to check if the word is less than two words like a,i, etc, I'm leaving them as it's as I think removing this connecting words will make the text look blatant that it has been attacked.
   b. I then define the functions which takes care of all the tasks. Like add_whitespace(word), swap_characters(w), substitute_char(w), delete_char(w), add_char(w).
      i. As their name suggest, the function works as follows respectively: adding whitespace, swapping characters, substitute characters, delete characters, adding characters.

   c. I then created an array of modification which stores the name of all the functions.
   d. After this I call the random which only picks modification with the 50% probability and if the probability comes more than 50% by the random function it returns the word itself.
      i. In the case probability is less than 50%, then I use the random choices function again to pick any one of the random modification techniques to apply the modification.
      ii. Post that I return the word.

def untargeted_attack(text):

1. This function has a small job of splitting the text or review it gets as an input as the example from a test data from the run_targeted_attack_on_test function.
2. The function than uses the random function and to use only in the time of the 40% of the cases and then calls the character_modify function for each one of the word and stores the word and appends it.
3. It then returns the modified text.

def adversarial_training(train_df,k):

1. This function takes in the training file and the argument k which states how many examples of the adversarial attack needs to be created for one single example. In this task we've worked with 5 values, 1,2,3,4 and 5.
2. The function iterates over each one of the examples and then class the untargeted_attack fcuntion for each one of the text examples.
3. The values are then appended along with the existing training data and labels to create a new file, which we save and subsequently return the data frame.

def evaluate_model(df, model, tfidf):

1. This function takes in the test data on which we evaluate the data on. This function is reused not only for the adversarial text, but also for the non-modified text examples. The function also takes the model as a parameter as we want to check the scores on both the non-attacked trained and adversarial trained model. The function also takes in the vectorizers for the adversarial or the non-adversarial trained model as they are used to transform the test text so that they can be evaluated to the perfection.
2. The function evaluates the model on different parameters like f1 score, accuracy, precision and recall.

def create_adversarial_test_set(test_df):

1. Like the name states, this function creates the adversarial test set for us, which we use to test our model on. This function is called later. It takes just the test data as the input and does it magic on that by creating adversarial test set by calling untargeted_attack function.

def evaluate_adversarial_experiments(train_df, test_df, orig_model, orig_tfidf, k_values=[1,2,3,4,5], runs=5):

1. This function is basically the place where all the function comes together and works. Like the argument names suggest, first we take the training data and test data as an input where both are unchanges as of now. The function also takes, original model(unattacked model) and unattacked vector embeddings. I also pass in the K values in the form of array in the function argument as my function takes in the k values as the argument which I described above. The runs are nothing but the number of times, the function is suppose to run. I could've defined run as a variable but went with the argument here.
2. The function calls the create_adversarial_test_set function to create the test data set for us to work on.
3. I then run the evaluate function on the adversarial attacked test set and the normal test to calculate how the model performs all in all.
4. Then I run a loop once for each one of the values in k(i.e the loop runs 5 time), which in turn runs the loop for 5 times for each value of k:
    a. What happens inside the loop is I store the adversarial result and original result with the different test set.
    b. Here I call the adversarial_training(train_df, k) which returns the attacked dataset for to create the attacked vectors for training the adversarial attack model.(I'm doing all of this inside the loop as I don't need the model later but then evaluate them).
    c. I then train the adversarial attack model from the embeddings we created above.
    d. I then evaluate the untrained adversarial model and trained adversarial model on the test set.
    e. I then store the above results in the array as the results will change when the loop goes ahead onto the next k.
5. After I print the average of result of the each value of k by each iteration of the loop.

I understand the above function can be a bit confusing, but I tried my best to explain it. If it gets trickier to understand it, comments might help you with it.

Analysis and Result:

The table below summarizes the evaluation metrics across the original model and all adversarially trained variants:

Model	Dataset	Accuracy	Precision	Recall	F1 Score
Original LR/Vectorizer	Original Test	0.750	0.734	0.782	0.757
Adversarial Test	0.721	0.726	0.711	0.718	
Adv. Trained LR/Vectorizer (k=1)	Original Test	0.750	0.736	0.780	0.758
Adversarial Test	0.732	0.729	0.737	0.733	
Adv. Trained LR/Vectorizer (k=2)	Original Test	0.756	0.741	0.786	0.763
Adversarial Test	0.733	0.734	0.731	0.733	
Adv. Trained LR/Vectorizer (k=3)	Original Test	0.758	0.743	0.788	0.765
Adversarial Test	0.738	0.738	0.737	0.737	
Adv. Trained LR/Vectorizer (k=4)	Original Test	0.757	0.741	0.791	0.765
Adversarial Test	0.742	0.742	0.743	0.743	
Adv. Trained LR/Vectorizer (k=5)	Original Test	0.759	0.743	0.792	0.767
Adversarial Test	0.743	0.745	0.740	0.742	

From the table above, it's clear that the original model, which was not adversarially trained, struggled slightly when evaluated on the adversarially perturbed test data. While the drop isn't extreme (0.750 → 0.721 in accuracy), it does reflect some vulnerability to these untargeted attacks. What's interesting is that even with only 40% of the words being modified, and many not targeting key decision points, the performance drop was consistent.

After introducing adversarial training, especially from k=1 to k=5, we begin to see gradual but consistent improvements in all metrics. For example, at k=5, the model not only performs slightly better on the original test set (Acc = 0.759, F1 = 0.767) but also shows noticeable resilience on the adversarial test set, scoring an accuracy of 0.743 and F1 of 0.742. These numbers reflect that the model is indeed learning to generalize better by being exposed to perturbed data during training.

That said, the improvement isn't drastic and that's likely due to several factors. First, since this is an untargeted attack with only 40% of the words being modified, many of the tokens crucial for prediction remain unchanged. On top of that, the way the word is perturbed is randomly chosen from five different modification strategies: adding whitespace, replacing characters, swapping characters, repeating characters, or deleting characters. For each selected word, only one of these five operations is randomly applied, and there is no guarantee that the specific transformation significantly affects the classifier's understanding of the token. This randomness reduces the likelihood that the perturbed word leads to a meaningful change in the vectorized input.

So, while adversarial training is helping, it doesn't fully mitigate the attack which isn't too surprising given the randomness of the attack method, the partial perturbation, and the limitations of the vectorizer. To improve the model's robustness further, I think the use of semantically richer embeddings like Word2Vec or GloVe could help in detecting and reacting to these character-level changes more meaningfully. Alternatively, using character-level or subword-level embeddings could provide better coverage against these kinds of attacks.

Overall, this part showed that adversarial training does improve robustness, but only marginally under the current setup. The results align with expectations based on the limited and randomized nature of the attacks and the structural limitations of TF-IDF. Still, the improvement across increasing values of k shows that the model is at least partially learning to deal with perturbations, and this opens a clear path for stronger modeling techniques in future iterations.