**Team:** Harshwardhan Yadav(PennID: 81440293), Prekshi Vyas(Penn ID: 26839769), Ivan Zhao (Penn ID: 40560187)

**Computer Vision (Deep Learning Pipeline)**

**Data preprocessing.** Describe in detail your data preprocessing, including:

- What software packages you used
- How you set up the training/validation/test sets
- Any data preprocessing you used

***Same as Project Milestone 2 - Details can be found in our previous extensive report.***

**Deep learning models.** Describe in detail how you trained your deep learning model(s), including:

- What software packages you used
  **PyTorch**, **torch.nn**: For components such as convolutional layers, activation functions (ReLU), max-pooling layers, and fully connected layers. And also for the loss which in our case is nn.CrossEntropyLoss().
  **torch.optim**: To define the optimizers.
- **The neural network architecture you used**
  After conducting experiments with various layers and their arrangements, **we have developed our custom CNN architecture as shown below (using torch.summary)**

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 32, 32]             896
              ReLU-2          [-1, 32, 32, 32]               0
            Conv2d-3          [-1, 64, 32, 32]          18,496
              ReLU-4          [-1, 64, 32, 32]               0
         MaxPool2d-5          [-1, 64, 16, 16]               0
            Conv2d-6         [-1, 128, 16, 16]          73,856
              ReLU-7         [-1, 128, 16, 16]               0
            Conv2d-8         [-1, 128, 16, 16]         147,584
              ReLU-9         [-1, 128, 16, 16]               0
        MaxPool2d-10           [-1, 128, 8, 8]               0
          Flatten-11                 [-1, 8192]               0
           Linear-12                 [-1, 1024]       8,389,632
             ReLU-13                 [-1, 1024]               0
           Linear-14                  [-1, 512]         524,800
             ReLU-15                  [-1, 512]               0
           Linear-16                   [-1, 10]           5,130
================================================================
Total params: 9,160,394
Trainable params: 9,160,394
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.77
Params size (MB): 34.94
Estimated Total Size (MB): 37.73
----------------------------------------------------------------
```

- **Values of key hyperparameters that you chose**

| Hyperparameter | Values |
|---|---|
| Optimizers | Adam |
| Learning Rate | [1e-4, 1e-3] |
| Betas (for Adam) | [(09, 0.999)] |
| Batch size | [16] |
| Epochs | [5, 8, 10] |
| Datashift | [True, False] |

- For hyperparameters that you varied, how did you vary them

  **Epochs**: the number of epochs we selected were 5, 8 and 10. We chose 10 as the upper limit of epochs because we found from preliminary results that 10 epochs reached high training accuracy. Thus, to avoid overfitting, we limited the epoches to 10 or less.
  **Learning Rates:** We tried different learning rates for all the models to cover a standard range which would give us information on what learning rates give a smooth curve and which end up providing fluctuations.
  **Optimizer:** We used Adam to assess model performance. Adam was chosen because in preliminary experimentation, Adam outperformed SGD in all cases. As such, we decided not to use SGD in our exploration.
  **Batch size**: We used a batch size of 16 in our data loader because in preliminary experimentation, we found 16 to give the best accuracy results.
  **Datashift**: We varied the input data such that they either had datashifts or not.

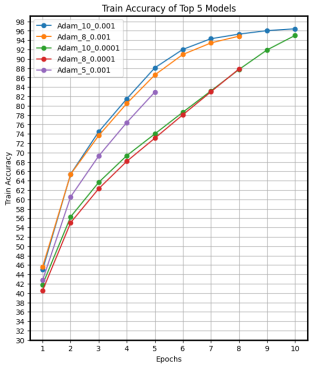
  **These hyperparameters in our grid-search gave a total of**
  **(1 optimizer) x (2 learning rates) x (1 beta values) x (1 batch size) x (3 epochs) x (2 datashifts) = 12 models.**

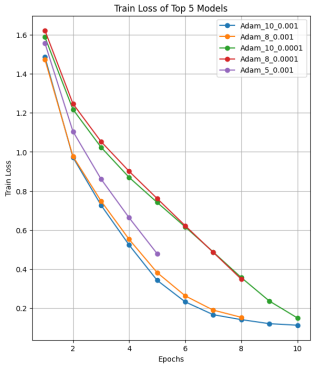
**Results and discussion.** Provide tables and plots showing your results, including:

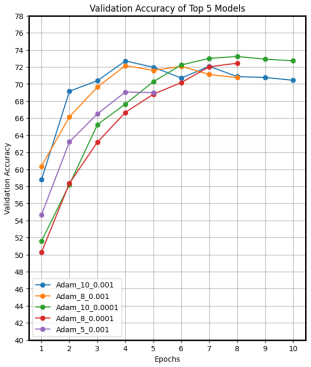
- **Training/validation/test results**

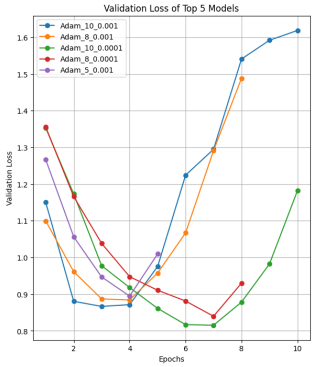
| Optizer | Datashift | Learning Rate | Epoch | f1-score |
|---|---|---|---|---|
| Adam | False | 0.001 | 10 | 0.690 |
| Adam | False | 0.001 | 8 | 0.686 |
| Adam | False | 0.0001 | 10 | 0.671 |
| Adam | False | 0.0001 | 8 | 0.647 |
| Adam | False | 0.001 | 5 | 0.640 |

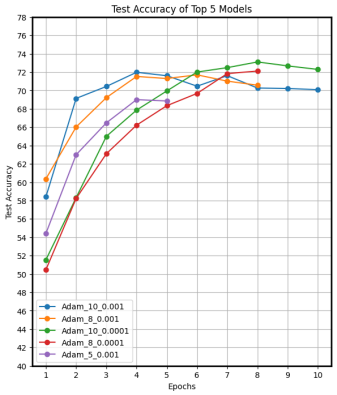
After evaluating the models in our grid-search method, the top 5 models were selected based on f1-score as seen in the table above. The following plots show the accuracy and loss over epochs for each model on the training, validation, and test sets respectively.

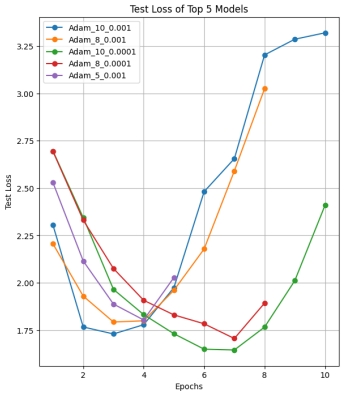



- **Results on how test performance varies with hyperparameter choices**
  **Epoch**: While the training loss decreases as epochs increase, we see that the validation and test loss decreases until around 4-8 epochs where it starts to rebound upward. This shows that with 4 to 8 epochs or greater, the models start to overfit the data. This could hinder test performance as a result.
  **Learning Rate**: Between the two learning rates, 1e-3 and 1e-4, 1e-4 gave the best test accuracy and lowest loss. A possible explanation is that Adam is sensitive to learning rates, so it performs better on a smaller rate. Also, a smaller learning rate means it would take more iterations to overfit, leading to a smaller loss.
  **Dataset Shift**: The top models with the best test performances were all trained on data that did not have dataset shift. This shows that training on data shifted inputs hurts test performance compared to unshifted data.

- **Comparison to performance of your traditional pipeline from Project Milestone 2**

The performance results of the best model in each respective pipeline. **(More in Results and Discussion section)**

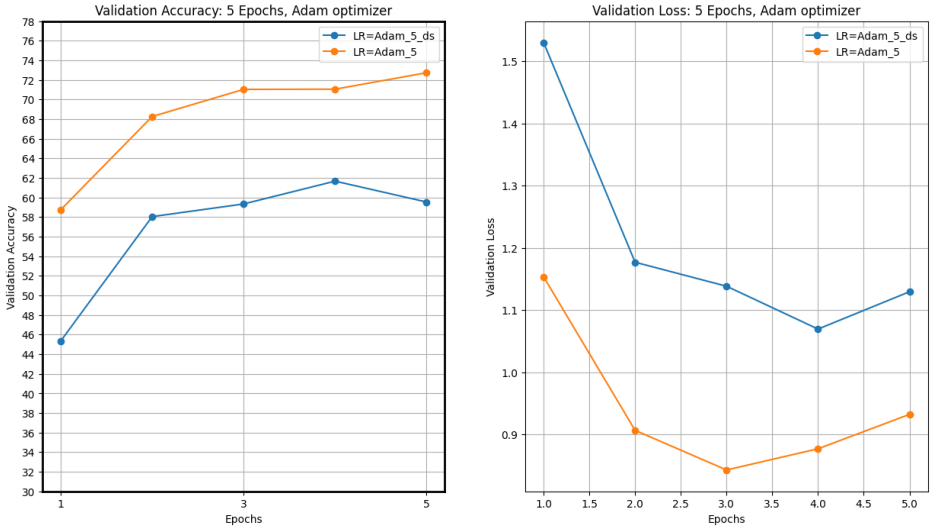
| | Convolutional Neural Network | Softmax Regression (Traditional) |
|---|---|---|
| F1-Score | 0.690 | 0.348 |
| Training Accuracy | 97% | 40% |
| Validation Accuracy | 72% | 39% |
| Test Accuracy | 72% | 39% |

- **Dataset shift experiment results for both traditional and deep learning pipeline**

  *Dataset shift experiment results have already been included in Milestone 2 for the traditional models.*
  *For Deep Learning pipeline :*
  According to this paper the preprocessing and data augmentation techniques can be found: https://arxiv.org/abs/1805.09501. After reviewing this: we have applied the following augmentation(random cropping):
  horizontal flips(50%) prob, zero padding(randomly between 0, 1, 2, 3) pixels, AutoAugment

***Results from DatasetShift***

Validation accuracy and loss from models trained without datasheets (orange line) versus validation accuracy and loss using models trained on datasheets (blue line). We denote the model trained with datashifts as “Adam_5_ds” and models without as “Adam_5”.




**In addition, provide discussion of the trends in your results, including the comparison to the traditional pipeline.**
Based on the data above, trends in our results include the following:

1. The models tend to learn better and perform at higher accuracies when no dataset shift is applied. One possible explanation is that the cropping of images in the datashifts removed too many important features. As a result, the model trained on data without datashifts is more complex and able to better predict the labels. This trend of datashifts worsening results was also observed in our results for our traditional pipeline (see milestone 2 documentation).
2. With 4-8 epochs we saw that the models started to overfit based on the increasing validation and test loss. In addition, at 10 epochs we saw that our model produced a training accuracy 97% compared to the 72% validation and test accuracy. Clearly this range of epochs between 4 to 10 the model gets high variance. For future tuning of the bias-variance tradeoff, we may consider finding the best number of epochs within this range.
3. In our results, a learning rate of 1e-4 gave the best training accuracy and lowest loss. One possible reason for this is that Adam is sensitive to learning rates, so a lower learning rate may work better for the optimizer. In addition, a smaller optimizer may slow down learning which means it takes longer for the model to overfit and increase variance.
4. Our CNN model is significantly outperforming our traditional pipeline (softmax regression) based on the results above. One possible explanation is that the softmax model was underfitted since because it was an one layer architecture. Meanwhile, our CNN had four convolutional layers in addition to multiple pooling and fully connected layers. Thus, our CNN is more complex which may lead to more accurate predictions. Another possibility is that softmax regression doesn’t account for the structure in images where CNN does take structure into account. Specifically, while our softmax regression model flattened the image into a 1-d vector and used that as the input, CNN uses convolutional filters that may reveal important spacial features. Thus, CNN is able to use useful features like location associations and spatial neighborhoods, which may lead to a more complex model and better predictions compared to softmax regression.

**Natural Language Processing (Deep Learning Pipeline)**

**Data preprocessing. Describe in detail your data preprocessing, including:**

- **What software packages you used**
  **re, string**: Working with regex and string operations, **Kaggle, zipfile**: Loading the Kaggle dataset and unzipping, **Pandas**: Reading the data from csv and storing it in pandas data frame, **nltk**: For data preprocessing tasks such as lemmatization and stop words removal, **bs4 (Beautiful Soup)**: Cleaning html tags from the data.

To prepare our inputs for the RNN-LSTM model we utilized the **torchtext** package.
**Torchtext** classes used:
**Field:** for processing TEXT and LABEL with **spacy** for tokenizing the TEXT fields using English tokenizer_language 'en_core_web_sm'. The vocabulary size is set to 20,000 most frequent words.
**TabularDataset** For creating parsed data sets.
**BucketIterator** For creating data loaders. The main advantage of using BucketIterator was that the similar length sentences get grouped together in a batch and it speeds up the training a lot as it reduces the padding overhead because the similar length sentences do not need to be padded much.

- **How you set up the training/validation/test sets**
  We divided the data as per the following ratios:
  train_ratio = 0.75, val_ratio = 0.125, test_ratio = 0.125 and random seed set to 42.
- **Any data preprocessing you used**

The following functions have been used to pre-process the data: Cleaning: Removing punctuation marks, non- alphanumeric characters, more than 1 consecutive spaces, trailing or leading spaces. Converting to lowercase, removing html tags. removing stop words, applying lemmatization, One hot encoding for the target values (sentiment = 1 if positive and 0 if negative) and Tokenization.

**Deep learning models.** Describe in detail how you trained your deep learning model(s), including:

- What software packages you used

Pytorch : **torch.nn** is used to build the entire network architecture. Moreover, matplotlib and sklearn libraries are used to evaluate the model predictions and visualize the loss and accuracy curves of the top models.

- **The neural network architecture you used**
  We created a SentimentAnalysis class for the DL pipeline. It consists of an embedding layer for mapping the TEXT tokens to vectors, with an option to initialize the embedding weights using pre-trained GloVe vectors. The embedded sequences are then processed through a bidirectional LSTM layer. We have then applied a dropout layer (to prevent overfitting), and the final hidden state is obtained by averaging the forward and backward hidden states. We are then passing this last hidden state through a linear layer with a ReLU activation function and dropout. Finally, the output is produced through a linear layer for classification. The architecture can be summarized as below(considering sequence length of 100 tokens) but we have trained different models with varying numbers of hidden units in the bidirectional LSTM layer which we make clear when we describe the hyperparameter choices.

```
==========================================================================================
Layer (type:depth-idx)                   Kernel Shape     Output Shape     Param #          Mult-Adds
==========================================================================================
├─Embedding: 1-1                         [50, 20002]      [-1, 100, 50]    1,000,100        1,000,100
├─LSTM: 1-2                              --               [-1, 100, 32]    8,704            8,448
│    └─weight_ih_l0                      [64, 50]
│    └─weight_hh_l0                      [64, 16]
│    └─weight_ih_l0_reverse              [64, 50]
│    └─weight_hh_l0_reverse              [64, 16]
├─Dropout: 1-3                           --               [-1, 100, 16]    --               --
├─Linear: 1-4                            [16, 16]         [-1, 16]         272              256
├─Dropout: 1-5                           --               [-1, 16]         --               --
├─Linear: 1-6                            [16, 2]          [-1, 2]          34               32
==========================================================================================
Total params: 1,009,110
Trainable params: 1,009,110
Non-trainable params: 0
Total mult-adds (M): 1.01
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 3.85
Estimated Total Size (MB): 3.91
==========================================================================================
```

- **Values of key hyperparameters that you chose**
  The key hyperparameters that we chose include the learning rates = [0.01, 0.001, 0.0005], epochs = [5, 15], embeddings type=[Glove, random initialization], Bi-LSTM hidden layer dimension = [8, 16], dropout probabilities=[0.2, 0.6], Optimizers = [Adam, SGD with Momentum of 0.9] and three types of datasets to train on including the original dataset, **dataset shift** applied to include **short sentences(of length 89 words** = the mean length of sentences - ⅓ * standard deviation of lengths of sentences) and dataset shift applied to include **long sentences(of length 150 words** = the mean length of sentences + ⅓ * standard deviation of lengths of sentences). In datashifts, we basically add "<pad>" tokens to match the sentence length. Moreover, some other hyperparameters that are fixed include the batch size=64, vocabulary size = 20000 and a weight decay of 1e-4 with the optimizers. These different choices of hyperparameters lead to training and testing of **288 different models**.
- **For hyperparameters that you varied, how did you vary them**
  For all the above mentioned hyperparameters, we firstly manually tuned the models and selected some hyperparameter choices for each hyperparameter that produced good results. Then, after the initial selection of some of the good choices for, we perform a grid search on the chosen hyperparameter values to obtain the top 5 models based on the F1 score performance on the validation sets. We very briefly describe our selection reasons. Firstly, **learning rates** of 0.01 and 0.001 seem to be the best rates to provide good results on a small number of epochs, learning rates less than 0.0005 slow down the training a lot. Generally, we obtain good enough models in 5 **epochs** with the Bi-LSTM model architecture but still some model choices seem to improve upon extensive training with low learning rates such as that of 0.0005 and so we test the models on two different choices for epochs. We tested the models with and without the **embeddings** initialized with Glove embeddings to test the effect of Glove embedding initialization on the model performance and learning capabilities. Initially, we tried the Bi-LSTM models that included a high number of units in the **hidden layer** such as 512 and 256 but the model showed overfitting and hence we kept on reducing the hidden layer's units till we got satisfactory results. We found out that for even 8 or 16 units in the hidden layer the model was still able to learn the task but below 8 it did not. Hence, we kept the units to

be either 8 or 16. **Dropout** was set to be either 0.2 or 0.6 as the model generally overfits the training data, hence we tried with these two different choices of dropout for regularization and dropout did control the overfitting, though not by much for some models. Finally, we keep three different dataset choices so as to train with **dataset shifts** and check whether the model still performed good on the original testing data.

**Results and discussion. Provide tables and plots showing your results, including:**

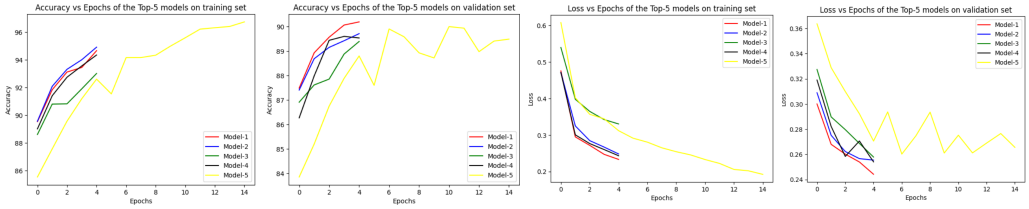
- **Training/validation/test results**

  **Out of the 288 different models** that we trained on, **we select the top-5 models based on the F1 score on the validation set** and then test those models on the test dataset. However, all the top-5 models seemed to give the best results for the original dataset, learning rate of 0.01 and also the Adam optimizer. Hence, to compare these top-5 models with the other hyperparameter choices, we also include the results of the best models for the hyperparameters that do not make it to top-5. The table below summarizes the model performance of the top-5 models(named Model-1, Model-2, …, Model-5), the best model with a learning rate of 0.001 and 0.0005, the best model with the SGD optimizer and the best models trained on the dataset after applying the dataset shifts and gives a ranking based on F1 scores. Also, we have shown the hyperparameter choices that produce these results.

| Model | Validation F1 | Test F1 | Validation Rank | Test Rank |
|---|---|---|---|---|
| Model-1 | 0.8982 | 0.8768 | 1 | 8 |
| Model-2 | 0.8939 | 0.8916 | 2 | 1 |
| Model-3 | 0.8935 | 0.8811 | 3 | 6 |
| Model-4 | 0.8925 | 0.8876 | 4 | 2 |
| Model-5 | 0.8920 | 0.8850 | 5 | 3 |
| Model-lr-0.001 | 0.8864 | 0.8833 | 6 | 5 |
| Model-lr-0.0005 | 0.8775 | 0.8769 | 7 | 7 |
| Model-SGD | 0.8738 | 0.8731 | 8 | 9 |
| Model-Short-Sentences | 0.8621 | 0.8842 | 9 | 4 |
| Model-Long-Sentences | 0.0 | 0.0 | 10 | 10 |

| Model | Glove? | Dataset | Optimizer | Learning rate | Dropout | Hidden_dim | Epochs |
|---|---|---|---|---|---|---|---|
| Model-1 | False | Original | Adam | 0.01 | 0.2 | 16 | 5 |
| Model-2 | True | Original | Adam | 0.01 | 0.2 | 8 | 5 |
| Model-3 | True | Original | Adam | 0.01 | 0.6 | 8 | 5 |
| Model-4 | True | Original | Adam | 0.01 | 0.2 | 16 | 5 |
| Model-5 | False | Original | Adam | 0.01 | 0.6 | 16 | 15 |
| Model-lr-0.001 | True | Original | Adam | 0.001 | 0.2 | 16 | 5 |
| Model-lr-0.0005 | True | Original | Adam | 0.0005 | 0.2 | 16 | 5 |
| Model-SGD | True | Original | SGD | 0.01 | 0.2 | 16 | 15 |
| Model-Short-Sentences | False | Short Sentences | Adam | 0.01 | 0.2 | 8 | 5 |
| Model-Long-Sentences | False | Long Sentences | Adam | 0.01 | 0.2 | 8 | 5 |

Moreover, the below figures show the loss and accuracy curves of the top-5 models while training on the training and validation datasets.




- **Results on how test performance varies with hyperparameter choices**

  For the results seen in the tables above, we can see that the higher learning rate of 0.01 produces better results as compared to the learning rates of 0.001 and 0.0005 with the same other hyperparameters(not necessarily the number of epochs), for this a comparison can be made between the Model-4 and the best models on learning rates 0.001 and 0.0005, and also all the best models tend to be achieving the best possible results on a learning rate of 0.01. The models seem to be not much different whether the embeddings are initialized using the Glove embeddings but still, we obtain the top-2 models on the test set that have their embeddings initialized via the Glove word vectors, with this we can say that initializing with pretrained embeddings then fine tuning does not provide a very significant boost to the accuracy of a model. Moreover, we can see that the best models are obtained when they are trained on the original training dataset. However, we can see that when trained on the datasets with dataset shift applied, the dataset of short sentences still produces competitive results on the original test set with the top models, whereas the dataset of long sentences tends to make the model learn nothing. With long sentences, the model architecture used does not learn anything and almost always predicts each sentence to be belonging to the same class, giving an F1 score of 0(see above and beyond section for more reasoning on this). With this, we can conclude that training the model on the original dataset or the dataset of short sentences seems to provide a good test performance. The optimizer Adam performs better than SGD+momentum optimizer in the majority of the cases. Generally, for many of the models out of the 288 models, the SGD optimizer learns too slowly and does not converge efficiently. Dropout ratios of 0.2 and 0.6 both seem to produce good enough test performances but still the majority of the models perform better when a dropout of just 0.2 is applied as it is easier for the models to learn with a lower regularization. With the choice of hidden dimensions, we can see that both the choices are able to learn, but with the choice of 16 hidden units, a higher proportion of models seem to produce good test performance, indicating that changing the hidden dimensions from 8 to 16 does not cause the models to overfit but instead increases the capacity of the model to generalize better. Finally, most of the models are able to learn the task within 5 epochs but for the Model-5 we can see from the loss curve that the loss keeps on decreasing with the epochs and benefits if trained on more epochs.

- **Comparison to performance of your traditional pipeline from Project Milestone 2**

  As described in the project milestone 2, the best result obtained from the Adaboost classifier was an F1 score of 0.74, whereas with the deep learning pipeline, the majority of the models achieved an F1 score of >0.86. This shows that the Bi-LSTM model is able to learn the semantics of the sentence better and is able to classify the sentiment of the sentences better as compared to the simple Adaboost classifier that basically

trained itself with a BoW mechanism and does not consider the structure of sentences. This can also be backed up by referring to the loss curves that the loss of the model in the traditional pipeline tends to be greater than 0.5 whereas the loss of the model used in the deep learning pipeline tends to be below 0.25 for the good models. If we also consider the feature extraction methodologies that were devised in project milestone 2, the hassle of extracting features is not required here. In project milestone 2, the dataset shifts required the feature extraction of embeddings and applying PCA whereas here we just need to pass the embeddings to the models and it learns those features automatically and learns even better. Hence, the time required for feature extraction is reduced quite significantly in deep learning pipeline but on the other hand the time it took to train the models also increased. Since, the time it takes to train an Adaboost model for 25 to 50 epochs is < what it took to train the Bi-LSTM for 10 epochs. This also creates a bottleneck to the different possibilities and tuning we could do with the deep learning pipeline.

- **Dataset shift experiment results for both traditional and deep learning pipeline**
  **The dataset shifts for the traditional pipeline have been explained in the project milestone 2 as well as have been included in the above and beyond section for reference**. On the deep learning pipeline, the dataset shifts applied(as explained in the hyperparameter choices section above) are of two types, 1) including short sentences only and 2) including long sentences only. The shifts in the deep learning pipeline are similar to the traditional pipeline. On the other hand, the traditional pipeline had dataset shifts that used long sentences and short sentences with the embeddings of the words either averaged or reduced in dimension with PCA concatenation which we named as X_train_LA, X_train_LCR, etc. respectively. With the dataset shifts in the traditional pipeline, we obtained the best test performance when we took long sentences and averaged their embeddings to represent the sentence as a 50-dimensional vector and got an F1 score of 0.75, whereas with short sentences, we got the best test performance with averaged embeddings producing an F1 score of 0.74. On the other hand, in the deep learning pipeline, the best test performance is obtained without any dataset shifts as shown in the result tables above with the F1 score of 0.89. On the other hand, the short sentences achieve an F1 score of 0.88 and the long sentences produce an F1 score of 0. Thus, we can say that the deep learning pipeline does achieve good test performance with short sentences but none with the long sentences. **We have also included the result table for the traditional pipeline after dataset shifts in the above and beyond section**. Thus, we can conclude that the dataset shifts where long sentences are given to the traditional pipeline(Adaboost Classifier) produce optimum results, whereas the dataset shifts when short sentences are used in deep learning pipeline(Bi-LSTM) produces the optimum results. However, with the deep learning pipeline it is still better not to apply shifts to the data and train on the original sentences. **Since, the given model does not learn anything on long sentences, we include another model's results on the longer sentences in the above and beyond section and give some reasoning as to why this might be the case.**

In addition, provide discussion of the trends in your results, including the comparison to the traditional pipeline.

**We have included the dataset shifts applied to and results of the traditional pipeline in the above and beyond section. They are the same as what we described in the project milestone 2.**

**The trends in our result and comparison to the traditional pipeline are as follows:**

a) The deep learning pipeline learns best when training on the original data without any dataset shifts applied whereas for the traditional pipeline, the best result was obtained with a dataset shift that trained the model on long sentence contexts.
b) Moreover, the model using Bi-LSTM is very powerful since it is able to learn the task of sentiment analysis with just 8 to 16 hidden units and as low as 5 epochs. Though it takes more time to train the model as compared to the traditional pipeline model AdaBoost.
c) Keeping aside the other hyperparameters, the Adam optimizer produces an F1 score and accuracy of >0.85 for almost all the models trained. However, without hyperparameter tuning, training a basic model causes overfitting as training accuracy reaches >97%. With SGD, the learning is very slow and we require more epochs to converge. With the hyperparameter tuning, we are able to nudge the accuracy/F1 score to be >0.88, giving a boost of 3% in accuracy and also keeping the training accuracy of the model to be <95% and thereby also reducing the overfitting with dropout and L2 regularization applied. With hyperparameter tuning, the best accuracy/F1 score of the AdaBoost model is around 75% in the traditional pipeline, implying the deep learning pipeline is more powerful as there is a significant rise of 14 to 15% in accuracy.
d) Because the time taken to train the models is high we select the hidden dimensions to be only 8 or 16 as they produce competitive results to higher numbers of hidden units like 256 or 512 but significantly boost training time. This can be compared to the traditional pipeline where we use a higher learning rate with a smaller number of weak learners to train on to obtain faster results.
e) The feature extraction of the Bi-LSTM is automatic but it also does have a drawback as we can see in our results that the model is unable to learn for longer sentences when it has to deal with many "<pad>" tokens. Whereas, when we manually extracted features using averaging or concatenation with PCA, even though the AdaBoost classifier does not consider the structure of the sentence it still produced the optimum results as compared to the Bi-LSTM which drastically failed on the longer sentences.
f) After testing for dataset shifts and from the results table above, we can conclude that the Bi-LSTM learns and performs well on the test dataset when trained on the original dataset or with the dataset shift where short sentences are used.

**Above and Beyond**

A) Here we mention the dataset shifts applied in the NLP for the traditional pipeline. This is the same as what we also described in project milestone 2.

**Dataset shifts:** We train the models on two formats of the dataset. The first includes short sentences and the second includes long sentences. The short sentences are formed by setting the length of each document in the dataset to be (M – (1/3) S), where M is the mean length of documents and S is the standard deviation of the length of documents in the training dataset. Whereas, the long sentences are formed by setting the length of each document in the dataset to be (M + (1/3) S). With this, the short documents are 89 words each and the long documents are 150 words each.

For feature engineering, we used the TF-IDF vectorization to analyze the words that are important in specific documents but not very common in the entire dataset. We used sklearn's TfidfVectorizer vectorizer for this with the minimum document frequency and maximum document frequency set to 0 and 1, respectively, to take into account all the words. N-gram range used was (1, 3) i.e., unigrams, bigrams and trigrams used to represent documents in the feature space. But this gave a feature size of 5807919 for each document. Since, our model is weak and also the feature dimension is too large we did not consider TF-IDF. Also, it takes considerable time and memory to train on this feature extraction.

Due to this we test two different methods of feature extraction:

1) Glove Embeddings with averaging: - Here, we take the 50-dimensional glove word vectors trained on 6 billion tokens to represent each word. Then, to find the representation of the document we average the embeddings of all the words to create an embedding of 50-dimensional vector for a single document. Hence, if the document has m words each of which is a 50-dimensional word vector, the document would also be represented by a 50-dimensional vector in the feature space which is obtained by the averaging of the m 50-dimensional word vectors. Hence, the training data after this method of feature extraction becomes a matrix of size (37500, 50), where 37500 is the number of documents and 50 is the feature representation of each document.
2) Glove Embeddings with concatenation + PCA: - Here, we represent each document by concatenating the word embeddings of each word in the document. For example, if the document has m words, then the document would be represented by a (m*50) dimensional word vector in the feature space that is formed by the concatenation of the word vectors of m words. This gave a training dataset of (37500, 4450) for short sentences whereas (37500, 7500) for long sentences.
   Also, since this would cause a very high dimensional feature space representation of the document, we apply PCA to reduce the dimension of each document's feature vector (originally 4450/7500-dimensional) to 500-dimensional vector.
   The test accuracy with and without PCA applied on the dataset was 0.71 and 0.73. We can see that the model still provides similar/good enough accuracy with the dataset after PCA is applied and the model is trained significantly faster as compared to when trained on a dataset without PCA. Hence, we decided to apply PCA when we represent each document as a concatenation of the word embeddings.

For applying the above steps on a document, we add special tokens called "<PAD>" tokens to the document to reach the desired number of words. The embedding of the <PAD> token is set to a zero vector. Also, if the document contains words that are not seen in the Glove embeddings, we also replace those words with the <PAD> token.

Hence, we create four combinations of datasets to train the models on, they are as below:

1) X_SA: - Original dataset X after considering **S**hort sentences + **A**veraging of word embeddings to represent each document. Hence, the training dataset is **X_train_SA**, validation dataset is **X_val_SA** and testing dataset is **X_test_SA**.
2) X_SCR: - Original dataset X after considering **S**hort sentences + (PCA **R**eduction after **C**oncatenation of words embeddings) to represent each document. Hence, the training dataset is **X_train_SCR**, validation dataset is **X_val_SCR** and testing dataset is **X_test_SCR**.
3) X_LA: - Original dataset X after considering **L**ong sentences + **A**veraging of word embeddings to represent each document. Hence, the training dataset is **X_train_LA**, validation dataset is **X_val_LA** and testing dataset is **X_test_LA**.

4) X_LCR: - Original dataset X after considering **L**ong sentences + (PCA **R**eduction after **C**oncatenation of word embeddings) to represent each document. Hence, the training dataset is **X_train_LCR**, validation dataset is **X_val_LCR** and testing dataset is **X_test_LCR**.

B) Below is the result table from project milestone 2 and is included for reference.

- **Table:**

| Model | Dataset | Optimizer | T | Learning rate (lr) | F1-Score | Rank (Based on F1-Score) |
|---|---|---|---|---|---|---|
| AdaBoost | X_train_LA | Adam | 25 | 0.1 | 0.7498 | 1 |
| | X_train_LA | Adam | 50 | 0.1 | 0.7494 | 2 |
| | X_train_LA | Adam | 5 | 0.1 | 0.7484 | 3 |
| | X_train_LA | Adam | 25 | 0.01 | 0.7416 | 4 |
| | X_train_SA | Adam | 50 | 0.1 | 0.7394 | 5 |
| | X_train_SA | SGD | 50 | 0.001 | 0.4524 | 51 |

**Now, we provide results of the deep learning pipeline on Long sentences.**

As we saw in the result section of the deep learning pipeline in the report, the model was not learning anything for long sentences and was giving an F1 score of zero. Basically, the model architecture that we used had less capacity to fit the data when longer sentences were given. The reason for this is that the model tends to underfit when a high number of tokens i.e.150 tokens per long sentence are given to the model. Moreover, in many sentences, since we fix the length of the sentence to be 150 tokens there are many “<pad>” tokens that the model has to deal with. Keeping in mind these reasons our choice of model architecture that performs well on the original dataset and the dataset of shorter sentences i.e. around 89 words, does not learn well when the number of tokens are high i.e. 150 tokens and does not deal well with potentially many “<pad>” tokens.

But, if we increase the capacity of the model then we see that the Bi-LSTM is able to learn the data. For this reason, we trained and tuned a model that had more capacity as compared to the model chosen in the deep learning pipeline and trained it for 100 epochs. The model seems to overfit the training data but still produces better results as compared to the original model architecture in the deep learning pipeline that basically limits itself to 50% or random accuracy.

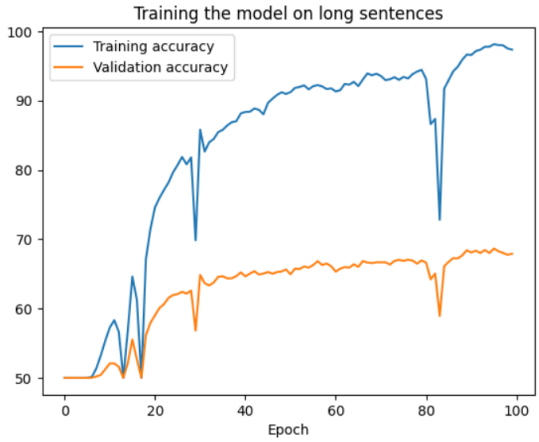
The model architecture of the new model specifically trained to be able to fit the long sentences is as below:

```
==========================================================================================================
Layer (type:depth-idx)                   Kernel Shape     Output Shape     Param #          Mult-Adds
==========================================================================================================
├─Embedding: 1-1                         [50, 20002]      [-1, 100, 50]    1,000,100        1,000,100
├─LSTM: 1-2                              --               [-1, 100, 32]    21,504           20,736
│    └─weight_ih_l0                      [64, 50]
│    └─weight_hh_l0                      [64, 16]
│    └─weight_ih_l0_reverse              [64, 50]
│    └─weight_hh_l0_reverse              [64, 16]
│    └─weight_ih_l1                      [64, 32]
│    └─weight_hh_l1                      [64, 16]
│    └─weight_ih_l1_reverse              [64, 32]
│    └─weight_hh_l1_reverse              [64, 16]
│    └─weight_ih_l2                      [64, 32]
│    └─weight_hh_l2                      [64, 16]
│    └─weight_ih_l2_reverse              [64, 32]
│    └─weight_hh_l2_reverse              [64, 16]
├─Dropout: 1-3                           --               [-1, 100, 16]    --               --
├─Linear: 1-4                            [16, 16]         [-1, 16]         272              256
├─Dropout: 1-5                           --               [-1, 16]         --               --
├─Linear: 1-6                            [16, 2]          [-1, 2]          34               32
==========================================================================================================
Total params: 1,021,910
Trainable params: 1,021,910
Non-trainable params: 0
Total mult-adds (M): 1.02
==========================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 3.90
Estimated Total Size (MB): 3.96
==========================================================================================================
```

Basically, the model used this time is the same as the model architecture we used in the deep learning pipeline but it is a stacked bidirectional LSTM model with 3 stacked layers of bidirectional LSTM. The hidden layers units have 16 nodes. Moreover, to speed up the training we train on a batch size of 256. The learning rate used is 0.01 keeping in mind that this learning rate produced the best results for the other models trained. We trained the LSTM model on the Adam optimizer with no weight decay and also set dropout to zero to avoid any regularization and promote the model to learn from the long sentences. The regularization is set to zero as the model finds it hard to fit the training data in the first place on longer sentences and hence we do not want to make this learning task even harder. Finally, the model is then trained for 100 epochs.

The results obtained are as below:

1) We show the training and validation accuracies per epoch in the below figure.




From this, we can see that the model overfits but still is able to learn something as opposed to the earlier case when it learned nothing from the training data.

2) Next, we show the model's accuracy and F1 score, along with the classification matrix in the below figure.

```
Test accuracy/f1: (69.02399444580078, 0.7192127473234129)
              precision    recall  f1-score   support

         0.0       0.74      0.58      0.65      3125
         1.0       0.66      0.80      0.72      3125

    accuracy                           0.69      6250
   macro avg       0.70      0.69      0.69      6250
weighted avg       0.70      0.69      0.69      6250
```

We can see that the model produces an F1 score of ~0.72 and an accuracy of 69% on the test dataset. This shows that the new model architecture does learn from longer sentences as well that have a large context and potentially many pad tokens per sentence.

From these results we deduce the following:

A) The longer the sentence, the more the need for a complex model architecture to capture the context. Especially, in our case where there are many extra pad tokens expected per sentence.

B) However, we cannot just increase the model capacity blindly as we see that the model tends to overfit as well. Moreover, we also need to apply regularization in an optimal manner so that the model does learn something. With this we conclude that building models that take in a long sentence or many numbers of tokens as a context is a very hard task.

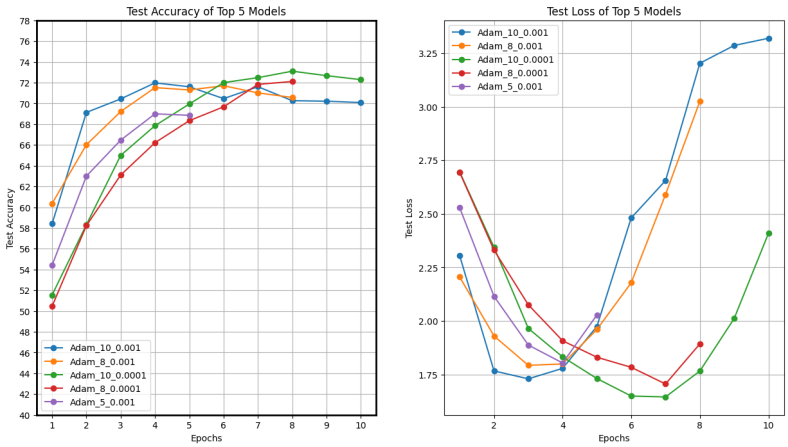
C) Moreover, the training time of the 3-layered Bi-LSTM that we used to learn the long sentences was around 25-30 minutes per 100 epochs. This means that training these RNN based models takes a very long time and this same fact makes hyperparameter tuning very difficult. One thing to note is that, the model that we train here is still a very small model as there are only a million parameters involved. Hence, to train on longer sentences, if we need to build more complex RNN based models than computational tractability also plays an important role.

D) Keeping in mind the results obtained on shorter sentences, we already get around 88% accuracy from those sentences and hence there is no strong reason to optimize hyperparameters for longer sentences which takes more time and needs better model architecture that are more complex and hard to come up with.

With this understanding, we conclude our analysis of the LSTM based model on long sentences and why the results differ so much when compared to other data shifts of shorter sentences and when compared to training on the original dataset.

B.) For the Computer Vision Pipeline first we tried several different preprocessing techniques for both traditional and deep learning pipelines to analyze which ones would yield the best results. These include:

- Sharpening the images but unnecessary portions of the image got sharpened, again distorting the images further.
- We also tried denoising using fastNIMeansDenoisingColored, but that takes the portions of the image that belong to the object of interest as noise. Hence, we only stick to the Gaussian blurring to remove noise. Gaussian blurring did not distort the image's object of interest.
- Apart from this, we also tried various data augmentation techniques after reading similar research works in the field [cited above in the "dataset shift experiment results" with details.]




Addressing Overfitting and Enhancing Generalization → Future Strategies for Model Improvement

Upon analyzing the test loss outcomes of our top 5 models, we see a trend where the loss begins to rise after approximately 4-8 epochs. In our future endeavors, we intend to enhance the model's generalization and mitigate overfitting by implementing techniques such as gradient clipping and batch normalization as these strategies will stabilize the training process by controlling the magnitude of gradients, and normalize inputs, contributing to improved model performance.