Report for Identifying and Categorizing Offensive Tweets

INL Subject

Artsiom Paliashchuk

s16550  PJWSTK

Table of Contents

[Requirements 2](#_Toc74013914)

[The idea of the model 2](#_Toc74013915)

[Preprocessing 2](#_Toc74013916)

[Model structure 6](#_Toc74013917)

[Task A 9](#_Toc74013918)

[Task B 17](#_Toc74013919)

[Conclusion, summary and possible improvements 25](#_Toc74013920)

# Requirements

In order to run this project, you don’t need to install anything else that is not present in the collab file. The order of running an executing is as follows:

1. Run 1 and 2 cells
2. In the newly created folder “Data” in the “content” directory add all files that are present in the task folder “IdentifyingCategorizingOffensive”
3. Run all other cells as the code will itself find and create the test, train and validation sets

# The idea of the model

The originally idea for the model is to create an interchangeable model (or two similar models) and use them to solve tasks A and B. Additionally, I tried to make the code and model in such way, that after creating the solution for task A , the solution for task B will be achieved by writing just some extra lines of code.

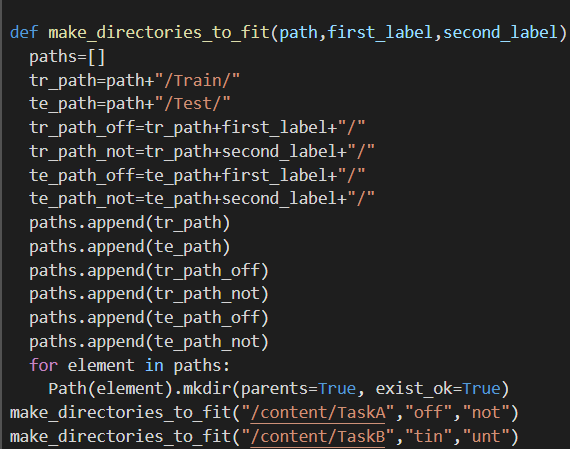
It should be stated that while the code or preprocessing was written by me, the original code for creating the structure of the model, vectorization and embedding were copied from the TensorFlow tutorial for text classification (<https://www.tensorflow.org/tutorials/keras/text_classification>)

My own experience with TensorFlow is at the lowest level (I have never written any neural network in it). My own experience with Python is not great either. As a result, I decided to use the tutorial and work alongside it to achieve results. Additionally, it should be mentioned that I was trying to follow the tutorial in such way, that my pipeline of the whole model creation will look as close to the tutorial’s one as possible. This leaded me to some very strange preprocessing that I had to do because my data format is very different from the one that is shown in the tutorial.

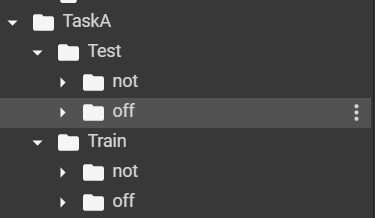
# Preprocessing

The preprocessing of the data consists of several steps. The first step is to make the data format of my data to be the same as in the tutorial’s data. To start with, my data was in tsv and csv formats in one (or several) text files. These files were consisting of tweet id, text message and labels and each file contained a lot of such records. The tutorial’s data was in format of txt file that contained simply the text. Additionally, the tutorial used the method tf.keras.preprocessing.text\_dataset\_from\_directory. This method allowed them to point to a specific directory and from it create testing, training and validation test sets. This method sounded good until I realized how it was implemented. According to it, this directory must have subdirectories (one subdirectory for each label to be categorized) and in each subdirectory must be text files (only in txt format). Each text file must contain exactly one record for the given label. For example, if there are 100 records for label A and 100 records for label B, then the whole directory must look like this:

A directory, in this directory there are two folders :”A” and “B” and in each of them there are 100 txt files that correspond to these labels. My original data was far from it and making my data fit such structure was a first problem. To solve it I had to manually create two folder (one for each task). After that I wrote a method that takes a path and two labels as arguments and simple create all the necessary folders. As a result, for method such as



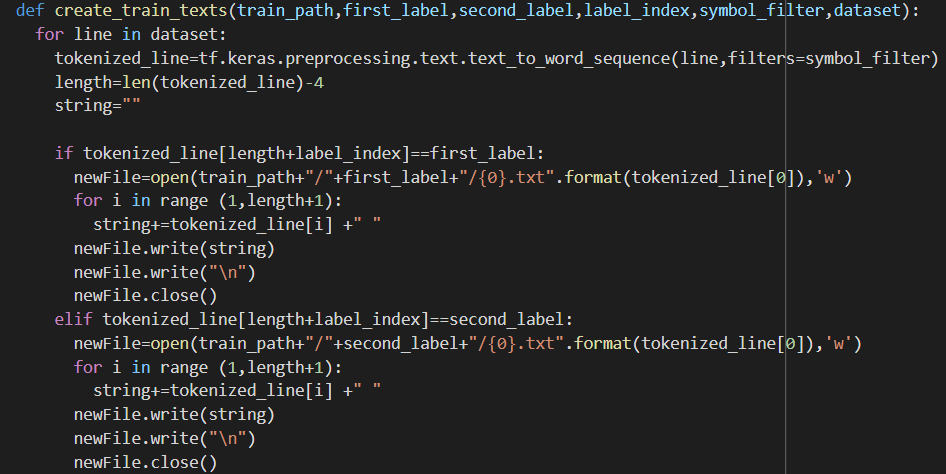
the output looked more like in tutorials, that is like this:



This architecture will fit the tutorials type of directory architecture, however, the folders were empty. Thus, the next step was to transform text data in my file into a bunch of txt files and place them in correct folders. Additionally, since we are looking into the data, this would be an ideal moment to process the data. My idea for processing the data is as follows:

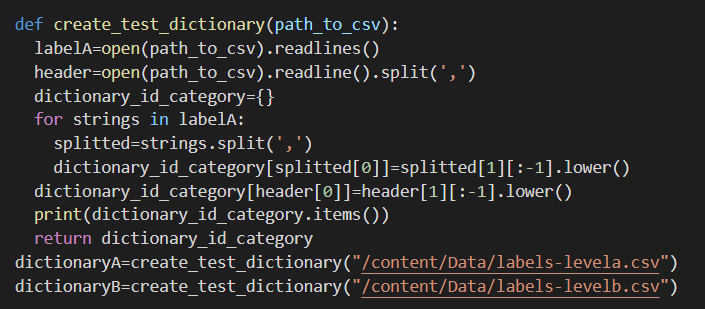
1. We read the data record by record (a record must contain ID of the tweet, text and 3 labels)
2. For each record that we read I have decided to put it into lowercase form and strip from some punctuation marks
3. For the same record we create a text file (.txt format) that is named with the ID of the tweet. The content of that file must be the processed text content of the tweet and this file must be assigned to such folder, that the label of this tweet corresponds to the name of the folder.

The code that does it is as follows:



For this method we path the main folder (root of these labeled subdirectories), two labels, number of label that we are interested in (for task A we are interested in the first label and for the task B we are interested in the second label), symbol\_filter (that is used to get rid of some punctuation marks) and the dataset from which we will read line by line. This piece of code works exactly according to the aforementioned steps so there is no need to explain line by line how does this code works. The only thing that should be explained is tf.keras.preprocessing.text.text\_to\_word\_sequence(line,filters=symbol\_filter)

This function works just as Python’s “split” function. It also splits the text by some delimiter (by default it is “ “). Additionally, it turns all lines in the text into lowercase form and remove all punctuation marks that are present in the filter. According to the tutorial the filter should be '!"#$%&()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n' but I have decided not to remove punctuation marks such # and @ because they are very often used in Twitter (# is used for hashtags (and one of the most famous hashtag in the dataset was #MAGA) and @ is used to address other Twitter accounts)). As a result of this method, the Train folder was filled correctly. The Test folder is filled using almost the same method. The only difference is that in Test dataset the correct label to modify weights is placed not inside the test dataset, but rather separately and is mapped by ID of the tweet that is present in the testing record. As a result, I had to firstly make a dictionary where ID is the key and label is the value. After that, during processing of testing record I looked up the ID of that record in the dictionary and extracted the label. Thus, I was able to correctly assign the testing record to the folder. Creating dictionary is simple:

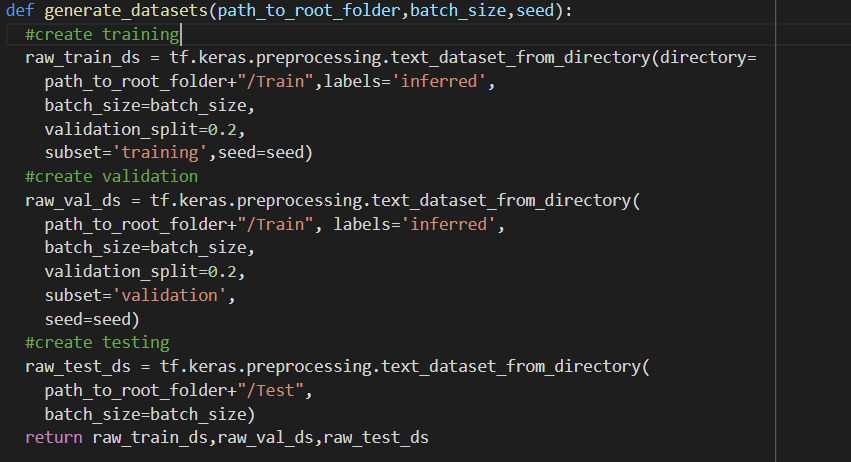


(I had to read the file twice as it was in .csv format, making the first line as a name of the columns and I wanted to extract them as well)

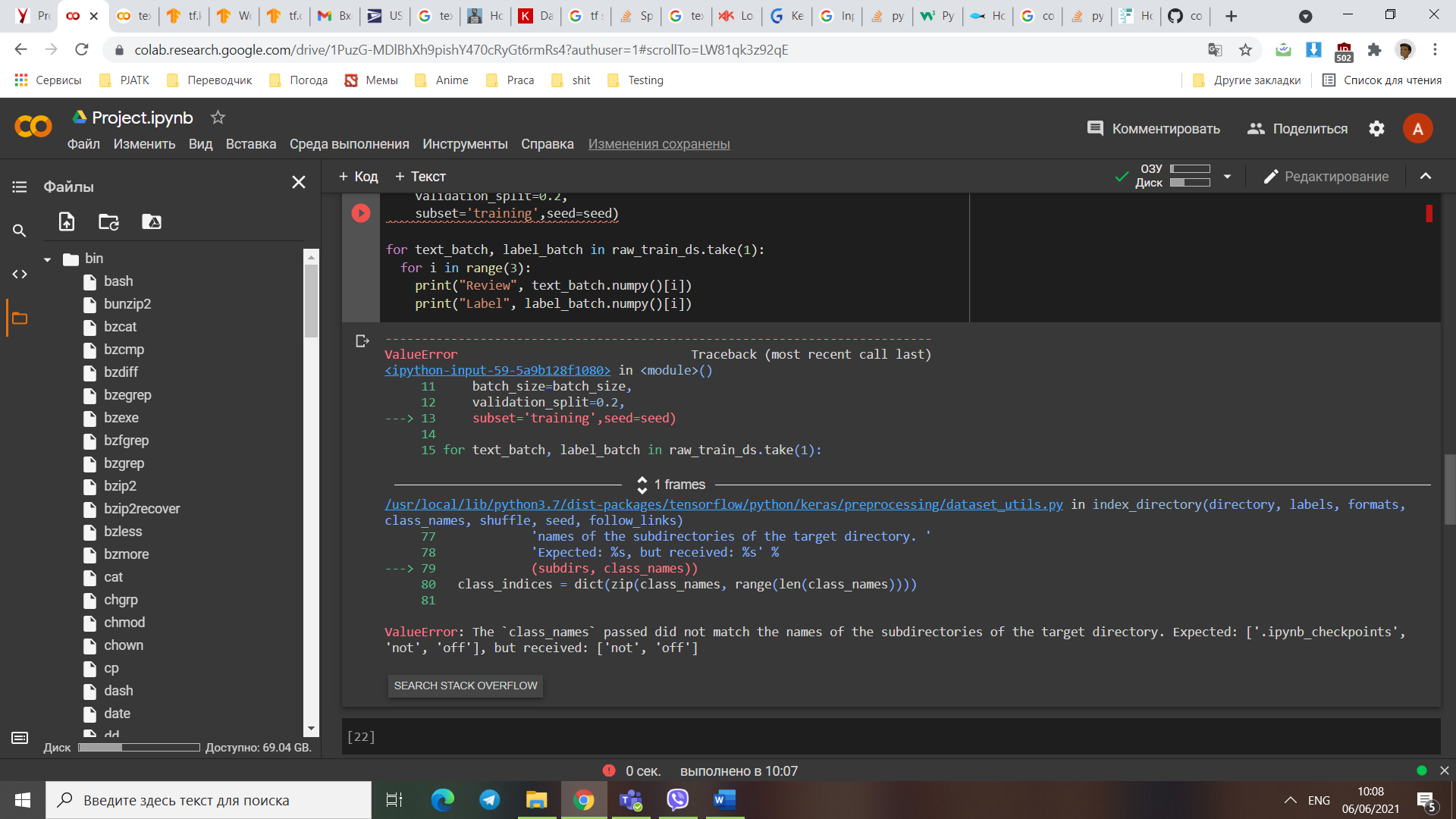
The last and the most important step was to create testing, training and validation datasets. All these weird preprocessing steps were done only to get here and make sure that it works correctly. Fortunately, for such folder structure as shown in the tutorial (and the one that I currently have), generating datasets is very simple. All that needs to be done is to call tf.keras.preprocessing.text\_dataset\_from\_directory with some parameters. Since I had testing and training data, making datasets from them was easy, however, I had no validation dataset. In order to create it, I’ve decided split it rom training data in proportion 80\20 (80 for training and 20 for validation) All this is done by assigning values to these two parameters:



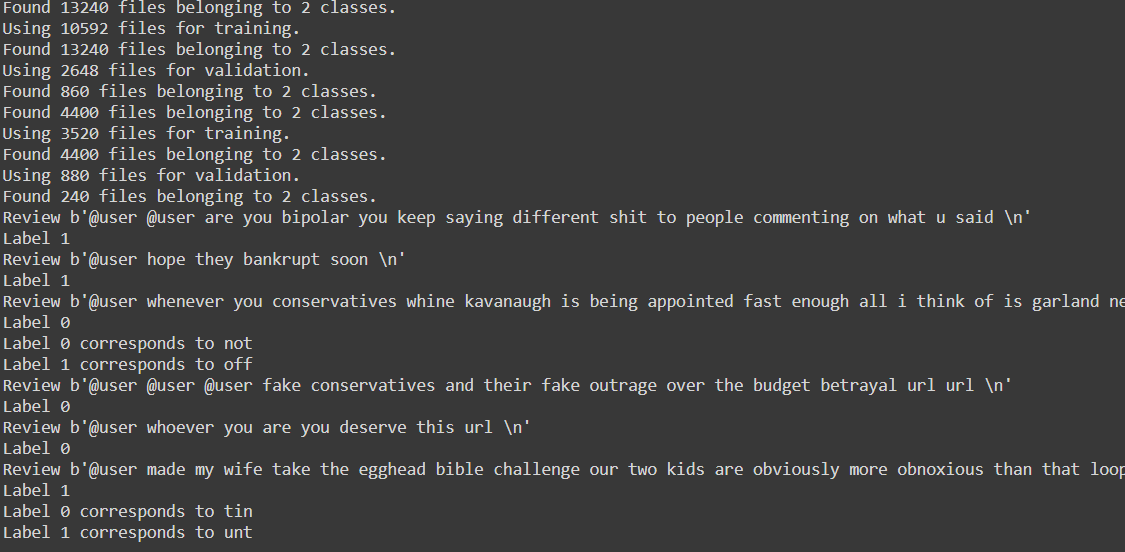
This will cause the validation subset to be of 20% of the size of the original testing data ). The whole method to create these three datasets is as follows:



Finally, I have decided to test these datasets and look on the labels of some random data from the dataset. On the next screenshots you will see some strange class “.ipynb\_checkpoints”. I didn’t know how to get rid from him. His presence ensures that I have 3 classes, and this makes binary classification impossible. I didn’t know how to get rid from it, but later it somehow disappeared.



And here are the results of these datasets (observe the 80\20 proportion)

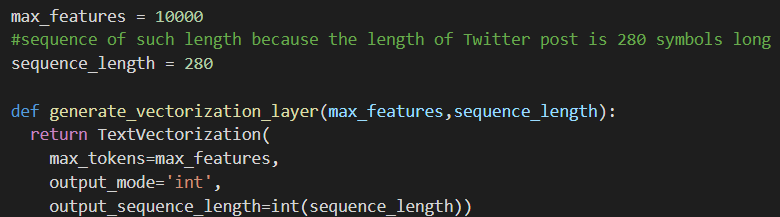


# Model structure

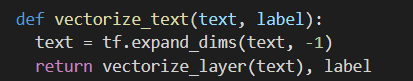
Since the datasets were created, now comes the main part where the model will be designed. Since I don’t have much experience with designing the models or the neural network, I’ve decided to use the model that is presented in the tutorials. Since the folder structures were the same as well as the fact that tutorial focuses on binary classification of sentence (almost the same task as mine), I’ve decided that it would be better to observe their model on my data and tune some parameters (where necessary and where possible). In general, the model has such structure:

1. Vectorization layer (it is outside the model but I also count it as a part of the model)
2. Embedding layer (it must be the first layer in the model, thus vectorization is outside the model)
3. Dropout layer (prevents overfitting)
4. GlobalAveragePooling1D (average pooling in order to be able to work with inputs of varying length)
5. Dropout layer
6. Dense layer (1) with just one activation unit (the main part of classification)

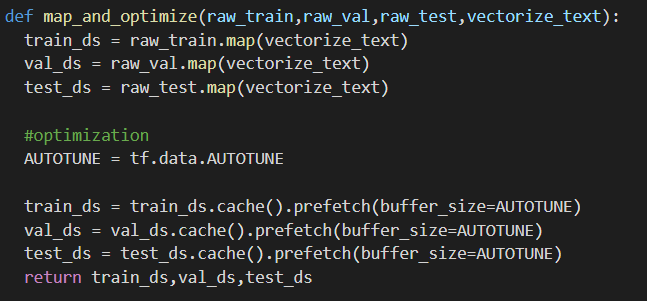
Since vectorization is outside of the model, it has to be done manually. For this, there is the code:



I have decided to use sequence length of 280 instead of 250 as 280 is the maximal length of Twitter post. After that comes the optimization and mapping part. In order to apply this vectorization, I had to map the dataset with a function:



The whole code for mapping and optimization is as follows:



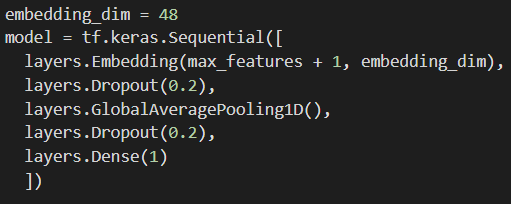
After the mapping was done, comes the time of optimization. Optimization is done using catch and prefetch. According to the documentation:

.cache() keeps data in memory after it's loaded off disk. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache, which is more efficient to read than many small files.

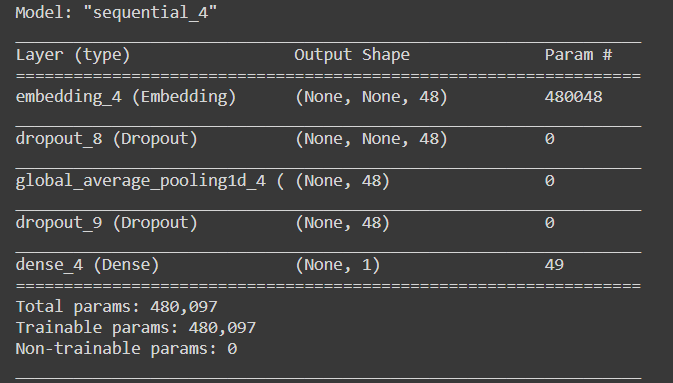
.prefetch() overlaps data preprocessing and model execution while training.

AUTOTUNE – parameter that may change over time, thus making the caching dynamic.

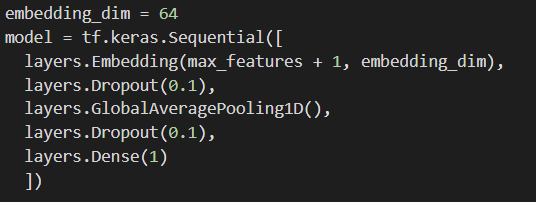
After vectorization is done, comes the part for the main model. The main model consists from all the layers except the vectorization one. The code for this model is as follows:

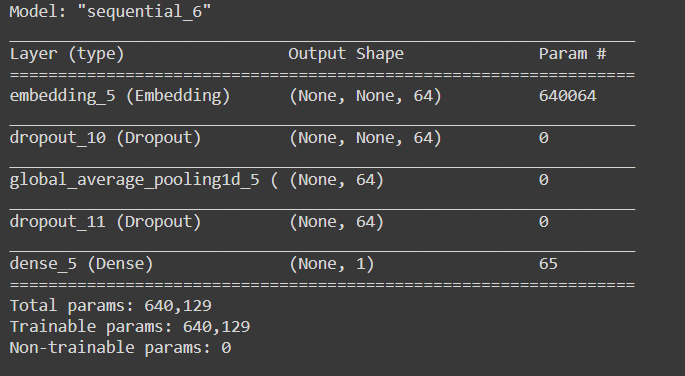


The values for Max features are taken at random (in my case it was 10,000) and embedding dimension was made to be divisible by two (whether it is divisible by two or not does not affect the performance). Values of the dropout were chosen through testing and observing the results. The summary for the model for the task A is this:



And for the task B is this:

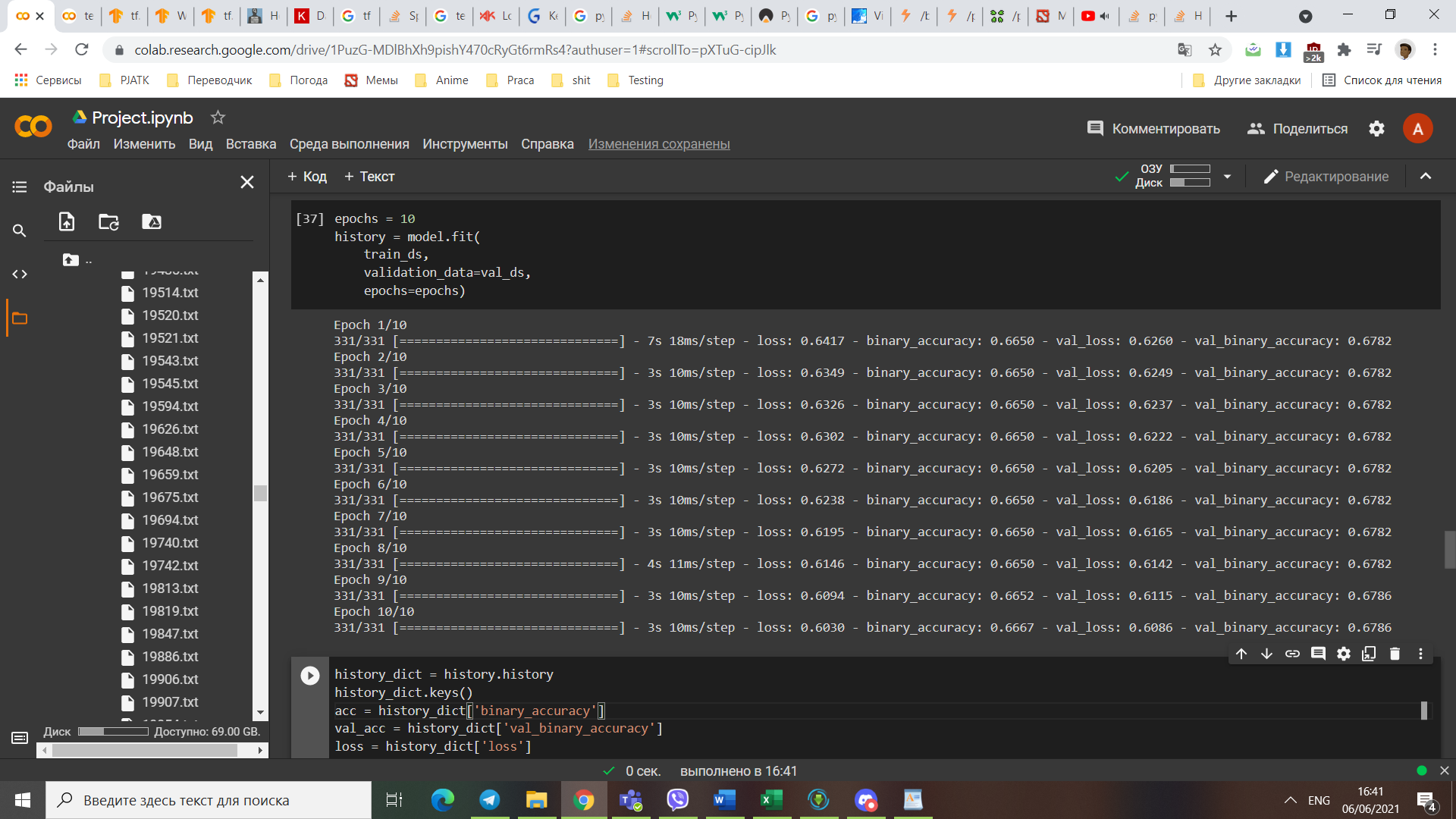


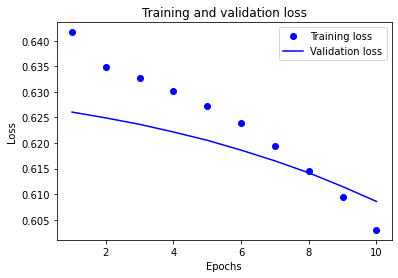


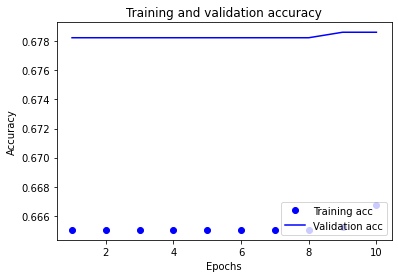
After the model is designed coma comes the part where we must choose the loss function the optimizer and the metrics for this model. Since our model is a binary classifier, the best loss function is BinaryCrossentropy (Computing the cross-entropy loss between true labels and predicted labels) with “From\_logits =True” (honestly, I did this because it was in the tutorial and in the documentation, it was ADVISED to set it to true). For the optimizer I have chosen “adam” but I haven’t observed any difference in using other optimizers. For metrics I’ve decided to use BinaryAccuracy with threshold of 0.5 (Calculating how often predictions match binary labels.) After all these values were inputted comes the part where we launch our model. Through testing it was discovered by me that the best number of epochs for the model is around 30.

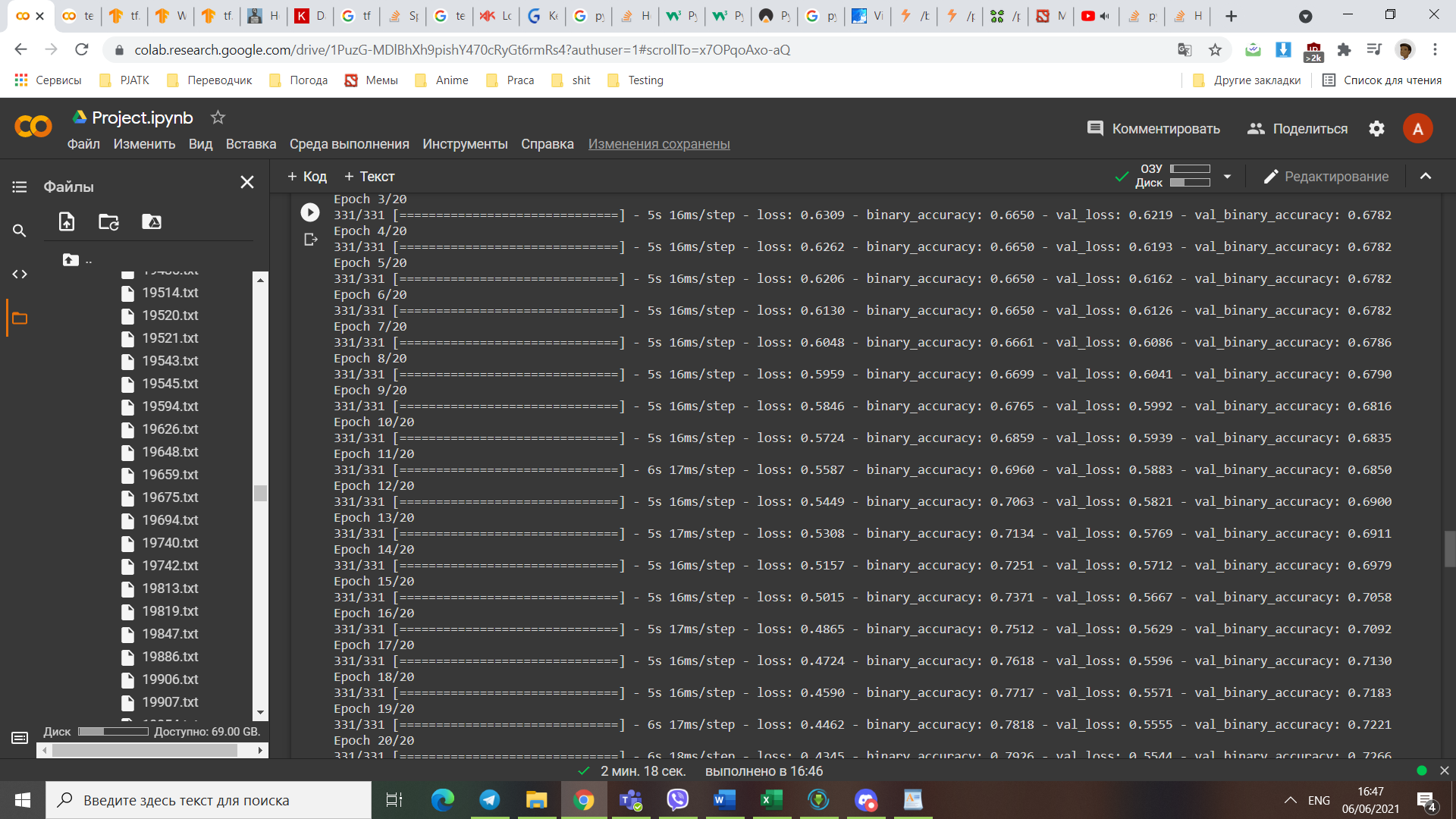
# Task A

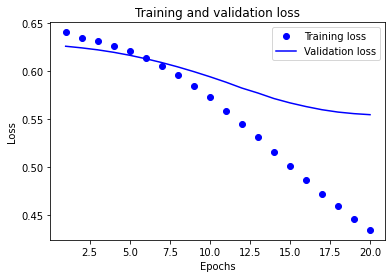
Down below you can find screenshots and performance graphs during different test runs.

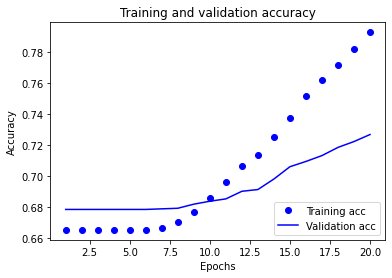




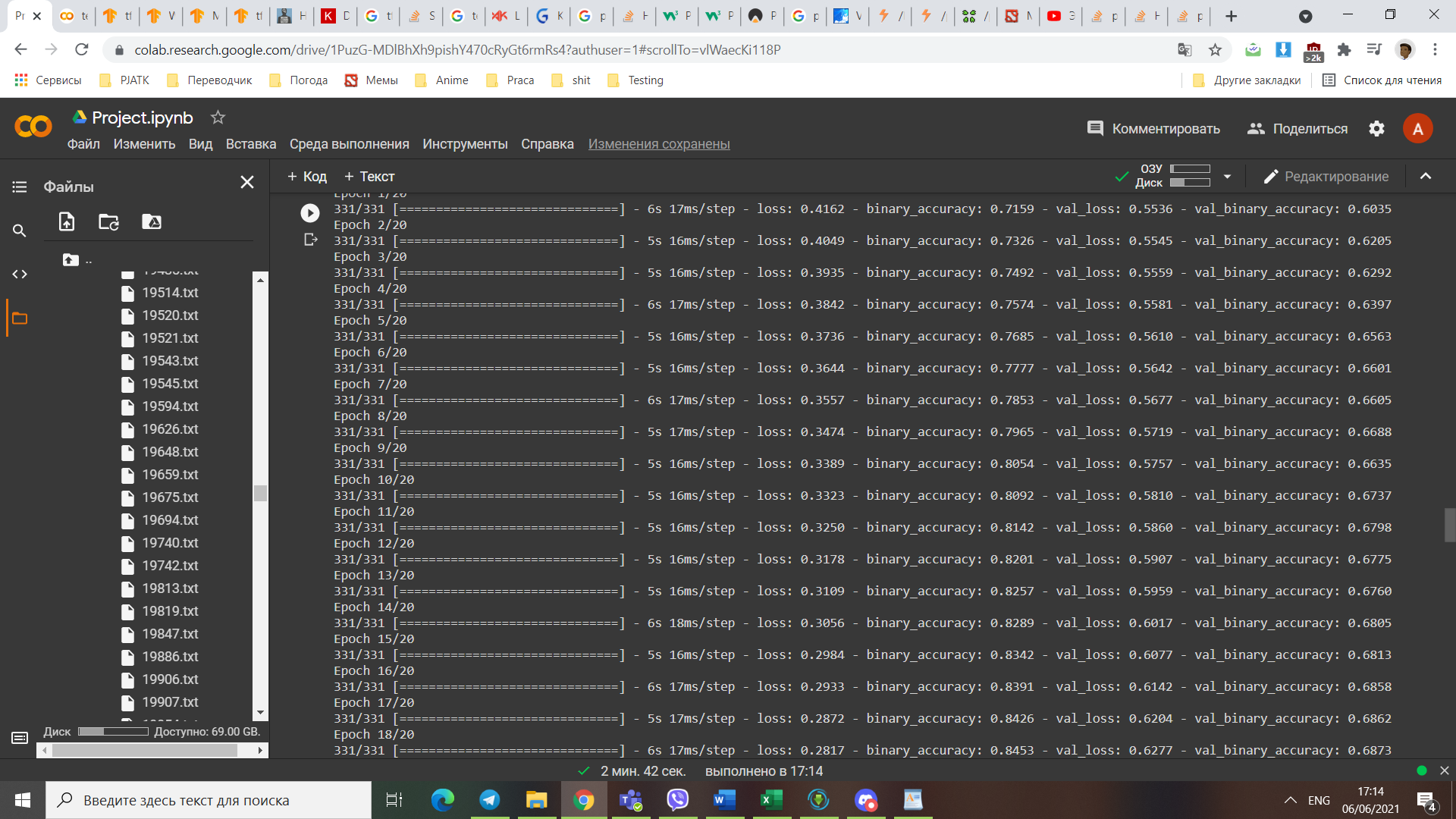


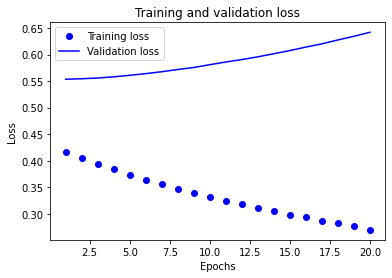


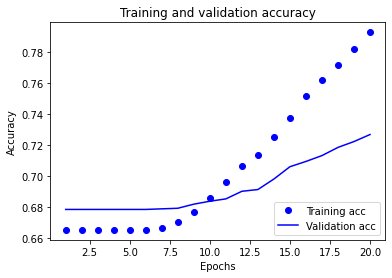




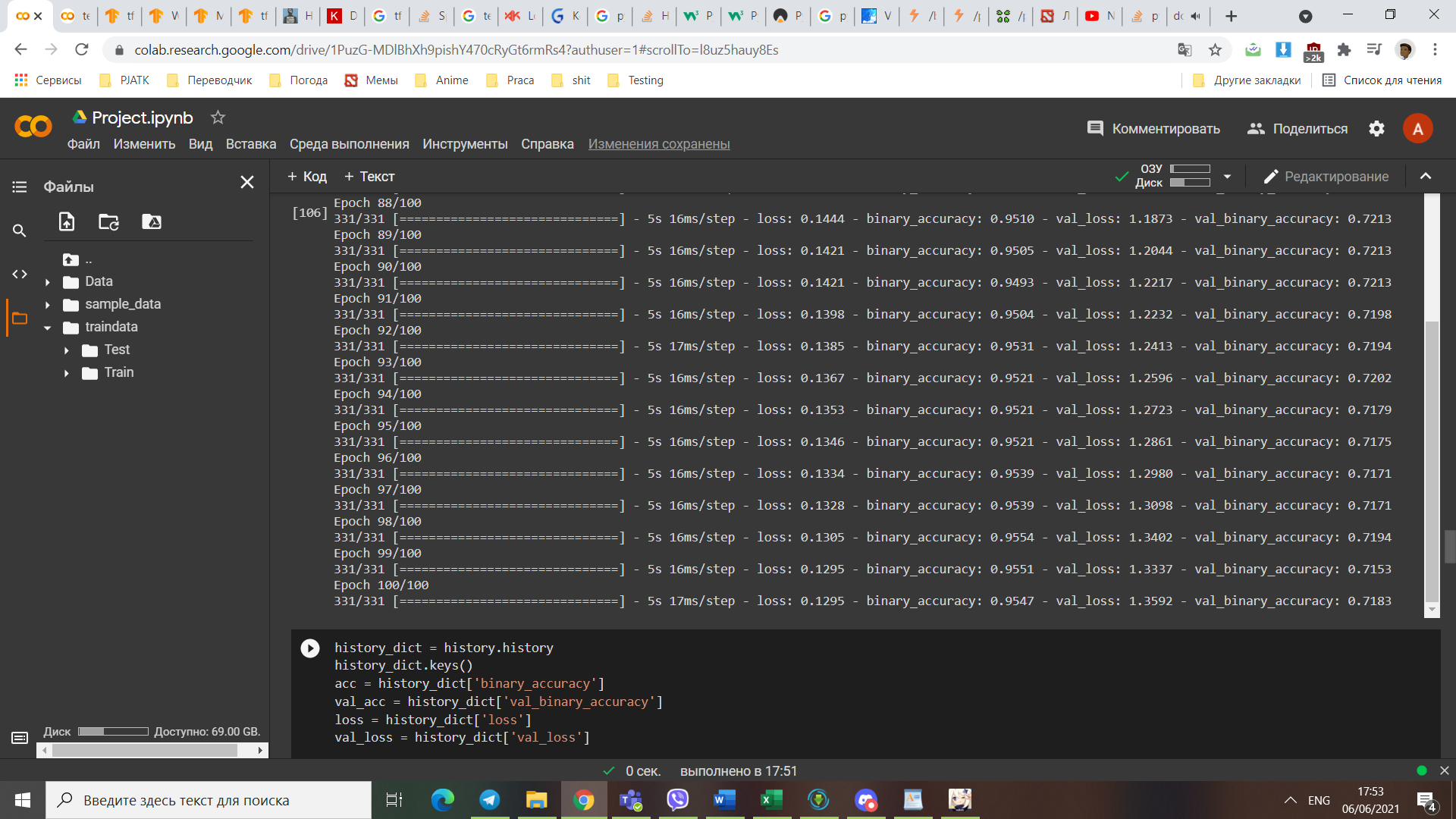
And here is the performance when “from\_logits =false”

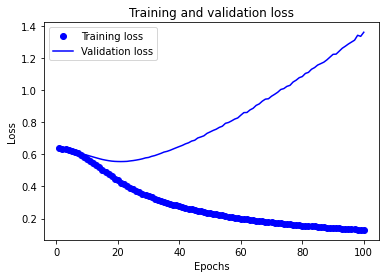


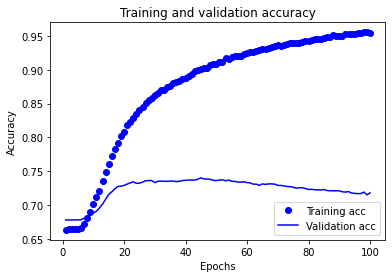




Here is what happened when I tried to put number of epochs up to 100. The overfitting was clearly seen







Using callback function that tracks some attribute did not give any result since the loss was always changing. As a result, I have decided to use the simplest method to find the best output: observe the graphs. On the graph it is seen that the peak accuracy for validation happens at around 27 epoch, so I can conclude that with given values, 27 epochs is the most optimal solution.

After performing many A/B tests on some values of the neural network (I changed only one attribute of NN and observed the results), I came to some numbers that I deed to be quite good for such tasks. These numbers are:  
max\_features =10000

sequence\_length=281

embedding\_dimension=48 (64 can also be used but it will take a little bit more time to train the model)

Dropout=0.2

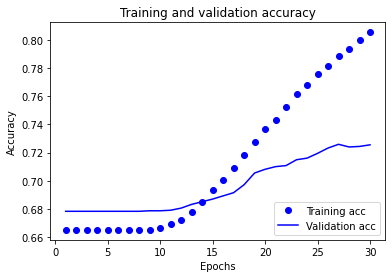
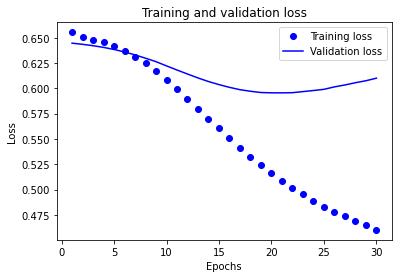
from\_logits=True (results are much better than if False)

label\_smoothing=0.15

metrics=BinaryAccuracy(threshold=0.5 (or 0.75 will also work fine))

epochs = 30 (or somewhere close to it)

The plots for accuracies and loses are as follows:



The question still remains: the model has 70% accuracy, but what does it mean? Does it mean that if the sentence is actually offensive, then it will be recognized as offensive with probability 70%. I tried to look into it by testing on some examples. The examples were:

"MAGA TRUMP MAGA TRUMP WON",

  "JOE BIDEN JOE BIDEN VOTE BLUE NO MATTER WHO",

  "NIGGER NIGGER NIGGER NIGGER NIGGER NIGGER NIGGER NIGGER NIGGER NIGGER",

  "Nigger is an offensive and derogatory word that is used to insulf all people of color (not only afro-americans) ",

  "Love is good, I love everyone and wish you all the best things in life",

  "Believe in Jesus and he will believe in you",

  "Democracy is one of the most important aspect of every US cityzen's life",

  "TRUMP",

  "BIDEN",

  "NIGGER JEW HOLOCAUST HITLER TRUMP PUTIN 9/11 ELECTION FRAUD  ALLAH",

  "NIGGER JEW HOLOCAUST HITLER TRUMP PUTIN 9/11 ELECTION FRAUD JESUS ALLAH Jesus Jesus",

As seen, these examples are clearly very offensive or very unoffensive. The results quite surprised me:

[0.24264237],

[0.39713567],

[0.84177446],

[0.2684738 ],

[0.04982293],

[0.44170934],

[0.07323289],

[0.27706003],

[0.28648967],

[0.27297127],

[0.5402326 ],

[0.7910484 ]

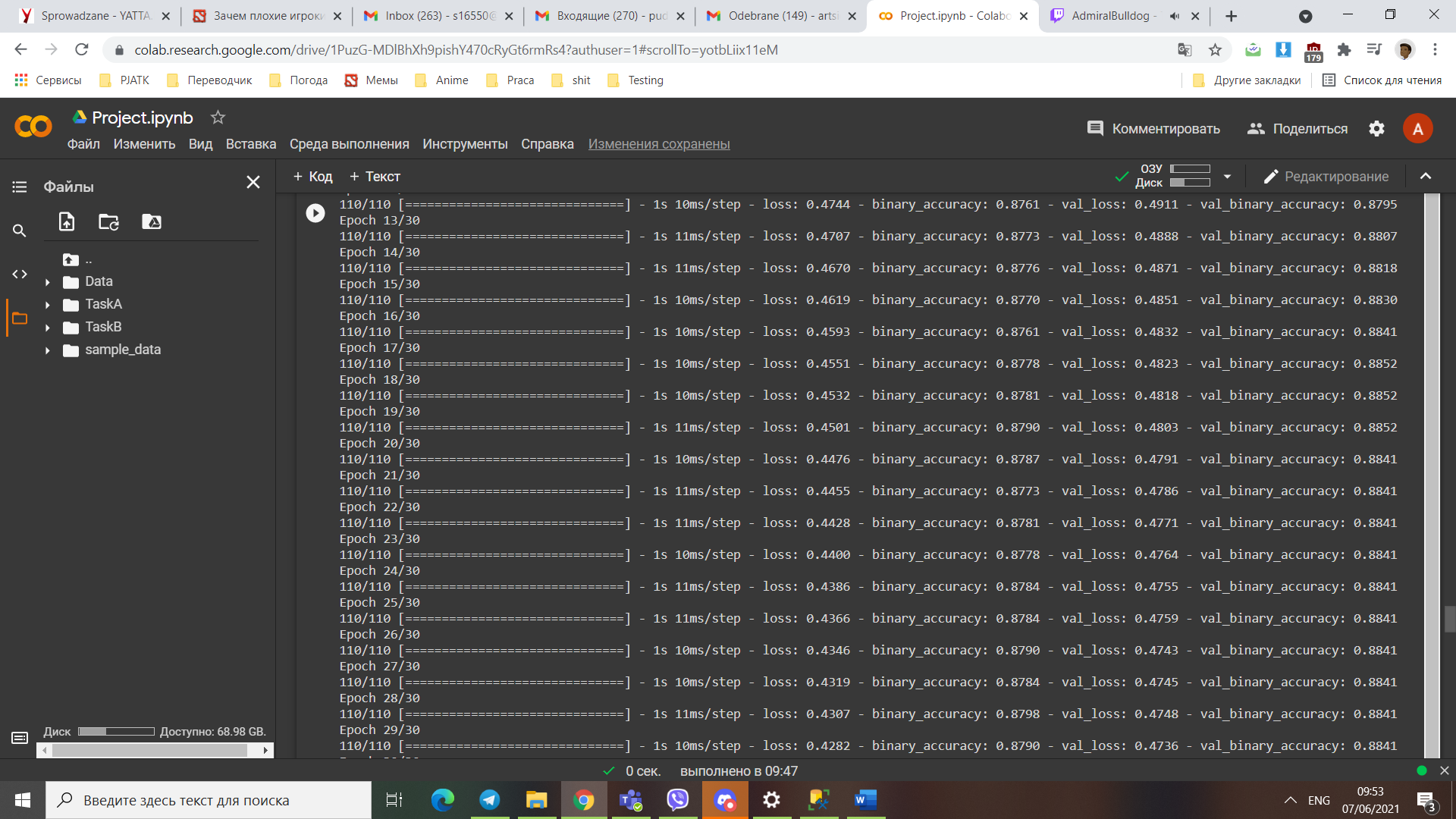
As seen, there are no such sentence that are clearly marked as offensive or not, however, we can try to interpret these numbers as “Degree of offence”, The closer the value to 0 is, the less offensive it is and the closer it is to 1, the harsher it is . In such case, the sentences that were containing some abstract good words (such as life, democracy, love, good, etc).

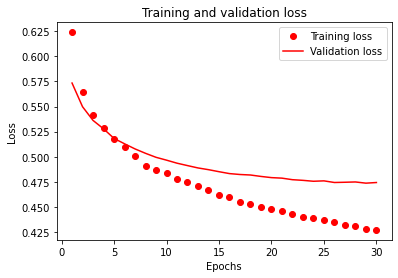
At the same time, surprisingly, sentences that were clearly offensive are ranked not that highly. At the same time, if we look at the last two values and sentences, we can see that they are only different in the word “Jesus”. This presence of such word makes the sentence more likely to be offensive. In other words, if the tweet contains the word “Jesus” then it is more likely to be offensive than the tweet without it. While these values don’t add up to 1, they can still give us an insight on what is offensive and what is not. Additionally, this model works perfect for A/B testing for offensiveness. For example, we can take any random sentence and change only one word and observe the result. By doing so we can see which word is more/less offensive. I believe that this is definitely a good use case for such model.

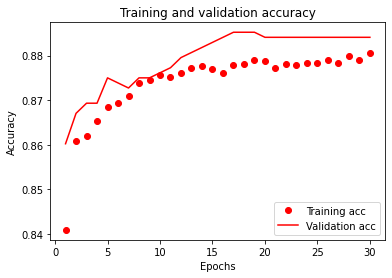
# Task B

The idea for the part B is somewhat similar. Since all of these tasks are classifications, I plan to use the exact same model and data processing for the part B and observe its performance. Finally, I would be able to plot both losses of these models on a graph and see whether the model can do the task and whether it makes sense to use such model.

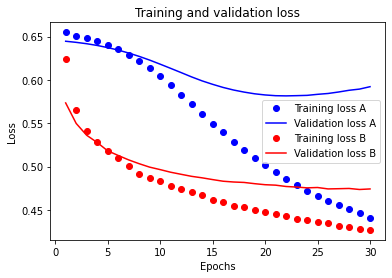
To my surprise, using the exact same model for this task proved to be quite interesting. For some reason, the performance of the same model in the B task performs better than in task A. There is a ~15% increase in accuracy and there seems to be no overfitting.





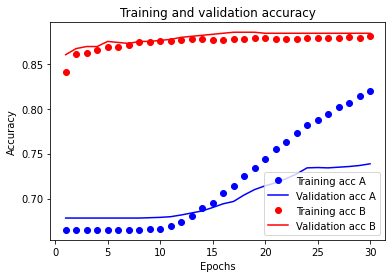


While these graphs show us quite good performance, it is important to evaluate the differences in performance for one model between two tasks:



As seen, the losses for the task B is much smaller and there seems to be no overfitting for the model for the task B. The loss for training data in task A decreases much faster then for task B.

The plot for accuracies comparisons looks as follows:



On this graph it is clearly seen the differences in accuracies. It is seen that at around 20 epoch the accuracy for task B stops increasing while the increase for accuracy for task A only becomes bigger. The same can be said about accuracy for training set. At epoch of 10 is stops to increase for task B and starts to rapidly increase for task A. from these graphs I can draw a conclusion that no further learning should be done with the given parameters as further learning for task A will cause overfitting, thus lowering the actual accuracy and for task B further learning will not improve the accuracy.

The last thing to test was the performance on some arbitrary strings where, I would assume, would be either targeted or not targeted insults. Unfortunately, the results were not that good. For the testing data that looks like this:

examples = [

  "You are a dumb retard, you are an asshole and you shoudl kill yourself, you are an idiot, you are a bigot",

  " dumb retard, an asshole and should kill oneself , an idiot, a bigot",

  "you are an IDIOT!",

  "IDIOT!",

  "You are the worst President in the hisotry of the US, I can't wait till you will be out of office. You are a nazi",

  "The worst   in the hisotry of the US, I can't wait till  will be out of office.  are a nazi",

  "You are like Hitler, dumb retarded nigger, KYS",

  "I am like Hitler, dumb retarded nigger, KYS",

  "He is like Hitler, dumb retarded nigger, KYS",

  "She is like Hitler, dumb retarded nigger, KYS",

  "Donald Trump is like Hitler, dumb retarded nigger, KYS",

  "Donald Trump is the worst Presiden in the history of the US",

  "Joe Biden is the worst Presiden in the history of the US",

  "Vladimir Putin is the worst Presiden in the history of Russia",

  "Donald Trump is the best President in the history of the US",

  " people,  white males are racists,  OF THEM, EVERY.SINGLE.ONE, they are bigots and dumb "

]

The results are not that good (although, the accuracy, apparently is quite high). The results are:

[0.47687775],

[0.70300573],

[0.2410636 ],

[0.27077246],

[0.09185991],

[0.15872964],

[0.22607476],

[0.3110091 ],

[0.22577226],

[0.2407417 ],

[0.20235085],

[0.13241485],

[0.21551019],

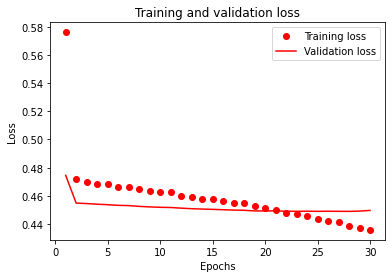
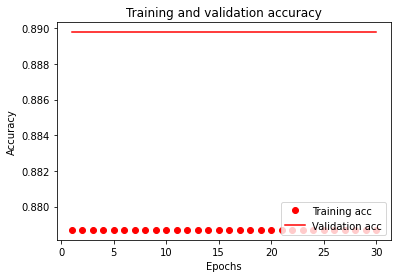
[0.22658485],

[0.05469838],

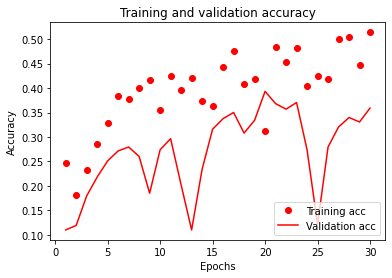
[0.23118821 ]

At first, I thought that removing pronouns would make the targeted insult into untargeted one, but I was wrong. Apparently, most of the data is categorized as a targeted insult and I can’t figure out what words effect the decision. The only thing that I DO managed to figure out is that if the sentence has some connection to the President then it will be definitely be categorized as a targeted insult.

After that I have learned about my mistake in the code. It seems that such results were happening because I forgot to reset the model and training for the task B was done on the basis of already trained model from task A. After resetting the model, there results were even worse.

It seems that this model from task A is completely useless for task B and must be changed. Tuning the parameters for the new model was a nightmare as results were very strange. Some graphs looked like this:

The results with tuning parameters are not impressive. The graphs look either like a flat line or like a random graph with ups and downs. After maty time spent tinkering with the parameters, I came to some parameters that work more or less well in this already no so good model. The parameters are:

max\_features =10000

sequence\_length=280

embedding\_dimension=64

Dropout=0.2

from\_logits=True (results are much better than if False)

label\_smoothing=0

metrics=BinaryAccuracy(threshold=0.5 )

epochs = 30 (or somewhere close to it)

The results for these values were as follows:

[0.05966979],

[0.16915986],

[0.1906769 ],

[0.23382622],

[0.04301471],

[0.07321331],

[0.16718039],

[0.2297903 ],

[0.15506902],

[0.17446208],

[0.14436722],

[0.1076771 ],

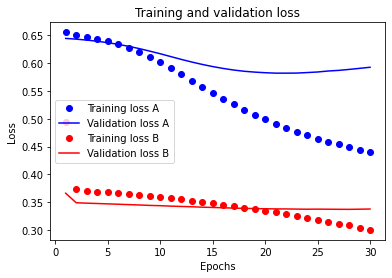
[0.13768905],

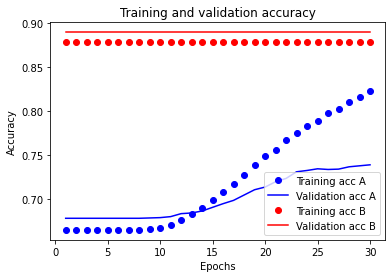
[0.13649312],

[0.11575702],

[0.1382077 ]

The final plots for accuracies and loses for both tasks are as follows:





From these values, we can draw a conclusion that this model is quite good for the first task and can quite well represent the offensiveness of the text. At the same time, the same model (yet even with different parameters) and its layer structure is not sufficient for adequate completion of the second task. It can find “definitely directed insults” and its performance can be observed only when comparing the results from directed and not directed insults. Another reason for some errors and results could lie in fact that the majority of the users of Twitter are from USA and this data was collected in 2019, just a year before President election. As a result, a lot of tweets were somehow connected with politics. Due to this, the neural network was quite sensitive to words that were concerning politics, such as President, MAGA, Donald Trump. Additionally, some abstract concepts and sensitive words were also affecting the results. Such, words as Love, Democracy, Good, Life makes the sentence less likely to be offensive while the word Jesus will make the sentence more likely to be offensive.

# Conclusion, summary and possible improvements

The way to improve the solution could lie in more different data. Additionally, the concept of “offensive” or “directed” is different to each person and is not recognized by the computer. Such, the word Nigger is offensive and will cause the sentence to be more offensive when the word is present in it. However, not all usages of that word can be considered offensive, however, the system doesn’t understand that and will make a neutral sentence to be slightly offensive. The only solution to this is the context of the words near this offensive word. This context can “tip the scale” and “lower” the offensiveness score.

Another thing that can be done to improve the solution (probably) is to make these tasks as a pipeline. As was mentioned previously, the necessity to classify task B comes only if in the task A the sentence is deemed to be an insult. In such case, the output of the task A should become the input of B. However, due to my lack of experience with these libraries I wasn’t able to perform this solution. Yet there is a drawback in such an approach. Since neural network can’t guarantee 100% accuracy, the errors in this approach will be propagated from the output of task A to the input of task B, which may lower the accuracy of the model in task B.

Third way to optimize it could lie in “decoupling” the tasks. As for now, both task A and B are done as binary classifiers. To remind, I’ve decided to do it on this premise based on this observation: in the task A the only two classes could be “Insult” and “Not Insult”, while in task B there are technically three classes: “Targeted”,” Non targeted”,” Null”. Since there are 3 classes, binary classification is not possible. My idea was to get rid of this “Null” label and not take it into account, thus making it possible to binary classify. This idea technically works because the null labels in task B only occur when in task A the label is “Not targeted”. Thus, every time in task A we get a result close to “Not targeted”, we can assume that for the task B this label is already “Null”. Thus, the only labels in question for the task B are “Targeted” and ” Non targeted”, thus, making it possible to solve task B using binary classification.

The idea for improvement is to treat this Null label as a separate label, thus making the data complete (as before I was throwing out the data that doesn’t fit the labels). Doing this will remove my ability to use binary classification for the Task B but may increase the performance of the task B since there would be more data (and even if the classifier would not recognize the null label as null, it will still learn on this data).

Another idea for improvent is the opposite of previously mentioned idea of decoupling. And it was actually presented by me here, but I discarded it as it was logically incorrect. This idea works on the premise that the solution for task B are the sentences that are the subset of task A. Technically, we can train the neural network firstly on the bigger subset (task A) and then use THE SAME neural network (without resetting it) to train and classify task B. Since the model was trained on the whole task A set, it solves the previously mentioned problem of null labels being excluded from the training for the task B

On the good side, my model is quite simple (although it is based on the model from tutorials from TensorFlow) and the learning process comes quite fast and or the task A the performance and end results are somewhat decent.

Another good point in my solution is usability. As seen from the code of the preprocessing, it may look strange how the preprocessing works (and I was surprised as well, but I did it in order to get the data to look like in the tutorial), however, clear code and some spent time of abstracting made it easy to easily convert the code from solution of task A to universal solution for this task. Technically, I would be able to create the solution for task C in just 5 minutes (using the same model from task A and B), however, task C is not solvable using binary classifier. If it was solvable, the I could just do it by adding one line of code in some sells and create a new model by copypasting the code.

Personally, this model may look not as something super great or something complicated, however, it is worth to mention that I have zero to none experience with TensorFlow and neural networks and my experience with Python is not that great either.

To sum up the advantages and disadvantages are as follows:

**Advantages:**

1. **Easy to use model that can be easily configured with additional binary classification tasks.**
2. **The model itself is quite simple and it learns very fast (learns up to 27 epochs)**
3. **Gives the ability to compare the same model when used for different tasks.**
4. **Using such weird preprocessing it is possible to easily or the user to define training, testing and validation datasets.**

**Disadvantages:**

1. **The results for task B are not that great due to dropping almost half of the data.**
2. **This model only works for binary classification (although, since my solution is based on TensorFlow example, it is possible to modify it to classify more than two labels)**
3. **Very strange preprocessing steps in order to make my data just like in the example from TensorFlow.**