Twitter Sentiment Analysis

Report

Mateusz Kwiatkowski, index 174159

Problem definition

The system will recognize tweets in form of text. It will be able to categorize if the tweet is positive, negative.

System input

Tweets in the form of text.

Each tweet will be either positive or negative.

System output

The system will return the class of the tweet. There will be 2 possible classes (labels):

1. Positive – tweet has a positive attitude
2. Negative – tweet has a negative attitude

System will output the label of each class (positive, negative). It will be either 1 (positive) or 0 (negative).

Example outputs for inputs:

1

0

Data description

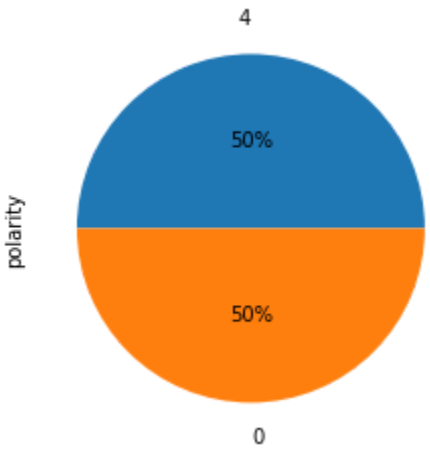
Dataset which can be downloaded from this link: http://help.sentiment140.com/for-students/ . Dataset contains 1,6 million records in the training dataset, file is in CSV format. Column definitions can be found in the given URL, but for our problem we are considering only “polarity” and “tweet” fields, where the first one is representing sentiment rating for the text which could be found under the “tweet” field.

Dataset content

Dataset contains over 1,6 million tweets with polarity rating, where 0 means that text found in the particular tweet has negative sentiment and 4 means that it’s positive.

polarity	tweet
0	my whole body feels itchy and like its on fire
4	I LOVE @Health4UandPets u guys r the best!!

In the plot below we can see that our dataset is balanced, exactly 50% of tweets are positive and 50% are negative.



Data representation

Text from each tweet in the dataset is represented by a vector, a function which computes vectors for the datasets is shown below.

```
vectorizer = CountVectorizer(
    analyzer='word',
    lowercase=False,
)
split1_features = vectorizer.fit_transform(
    split1_data
)
split1_features_nd = split1_features.toarray()
```

Data preprocessing:

```
def preprocess_tweet_text(tweet):
    # text to lowercase
    tweet = tweet.lower()

    # remove urls
    tweet = re.sub(r"http\S+|www\S+|https\S+", "", tweet, flags=re.MULTILINE)

    # remove punctuations
    tweet = tweet.translate(str.maketrans("", "", string.punctuation))

    # remove @ references and #
    tweet = re.sub(r'\@\w+|\#', "", tweet)

    # remove stopwords
    tweet_tokens = word_tokenize(tweet)
    filtered_words = [word for word in tweet_tokens if word not in stop_words]

    # stemming
    ps = PorterStemmer()
    stemmed_words = [ps.stem(w) for w in filtered_words]

    # lemmatizing
    lemmatizer = WordNetLemmatizer()
    lemma_words = [lemmatizer.lemmatize(w, pos='a') for w in stemmed_words]

    return " ".join(lemma_words)
```

I formatted all tweets to lowercase, removed urls, punctuations, @ and # references, stopwords. I also performed stemming and lemmatization.

Data splits

The whole dataset was split into training, validation and testing subsets. Three different such splits were created.

Split 1

```
split1_features = vectorizer.fit_transform(
    split1_data
)
split1_features_nd = split1_features.toarray()

split1_X_train, split1_X_test, split1_y_train, split1_y_test = train_test_split(
    split1_features_nd,
    split1_data_labels,
    train_size=0.90,
    random_state=1234
)

split1_X_train, split1_X_val, split1_y_train, split1_y_val = train_test_split(
    split1_X_train,
    split1_y_train,
    train_size=0.89,
    random_state=1234
)

print("split1 done")
```

First split was made by splitting data randomly by 80/10/10 proportions for the train/test/validation dataset. No preprocessing on that split was made.

Split 2

Second split was made similar to the first one, but with preprocessing.

Split 3

Split 3 was the same as split 2 but VAL was added to the TRAIN set.

```
split3_data = []
split3_data_labels = []
# Data preprocessing
for i in range(len(split1_data)):
    split3_data.append(preprocess_tweet_text(split1_data[i]))
    split3_data_labels.append(split1_data_labels[i])
print("Preprocessed")

split3_features = vectorizer.fit_transform(
    split3_data
)
split3_features_nd = split3_features.toarray()

split3_X_train, split3_X_test, split3_y_train, split3_y_test = train_test_split(
    split3_features_nd,
    split3_data_labels,
    train_size=0.80,
    random_state=1234
)

split2_X_train = split3_X_train.copy()
split2_X_test = split3_X_test.copy()
split2_y_train = split3_y_train.copy()
split2_y_test = split3_y_test.copy()

split2_X_train, split2_X_val, split2_y_train, split2_y_val = train_test_split(
    split2_X_train,
    split2_y_train,
    train_size=0.89,
    random_state=1234
)

split3_X_val = split2_X_val.copy()
split3_y_val = split2_y_val.copy()
```

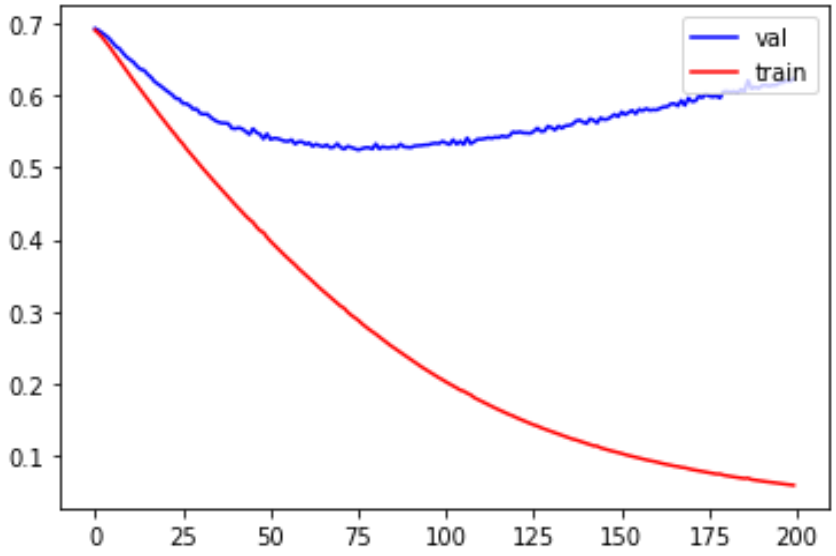
Type of problem and kind of training

I decided to go with the MLP model to check whether a tweet is positive or negative.

```
n_words = split1_X_test.shape[1]
# define network
model1 = Sequential()
model1.add(Dense(50, input_shape=(n_words,), activation='relu'))
model1.add(Dense(1, activation='sigmoid'))
```

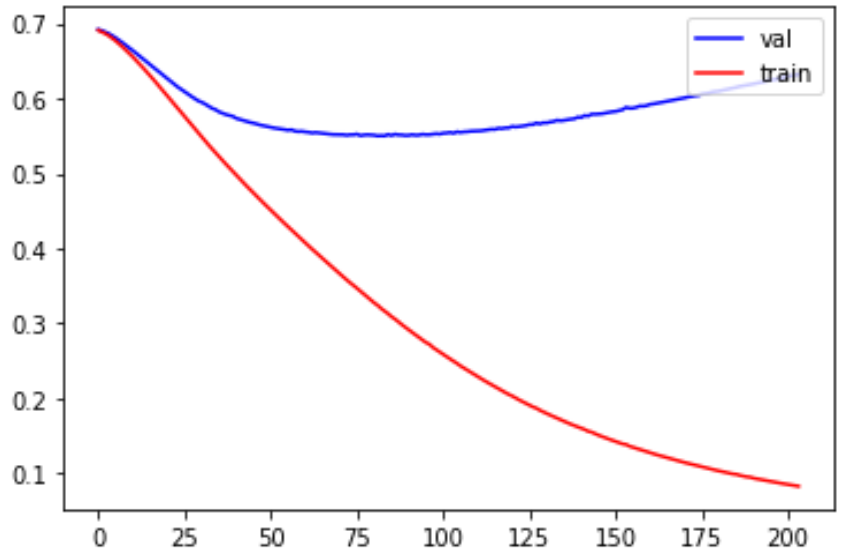
```
# compile network
model1.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy', f1_m, precision_m, recall_m])
```

Training on Split 1 dataset



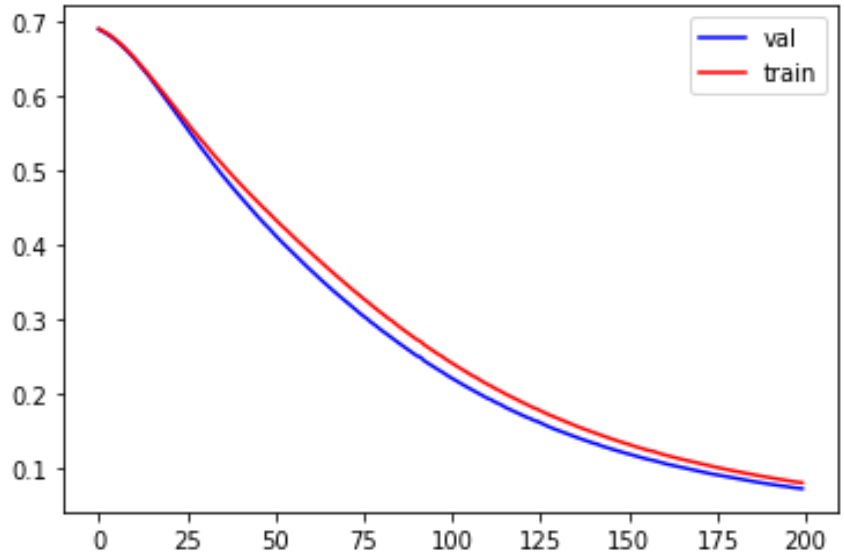
Execution time: 443s

Training on Split 2 dataset



Execution time: 420s

Training on Split 3 dataset



Execution time: 413s

Conclusion: Loss of validation in split 1 and split 2 was starting to rise because data is very difficult and hard to distinguish for the model. In split 3 validation was decreasing to a good

value because validation dataset was included in training dataset. We can assume that the model is in the best state in the 50 to 75 epoch.

I did classification problem so metrics that I used are: accuracy, precision, recall, F1 score.

Accuracy = correct/all

It is a good metric but it is bad for an imbalanced dataset because even then accuracy could be very good.

Precision = correct/all returned (tp / tp + fp)

Recall = correct/all relevant (tp / tp + fn)

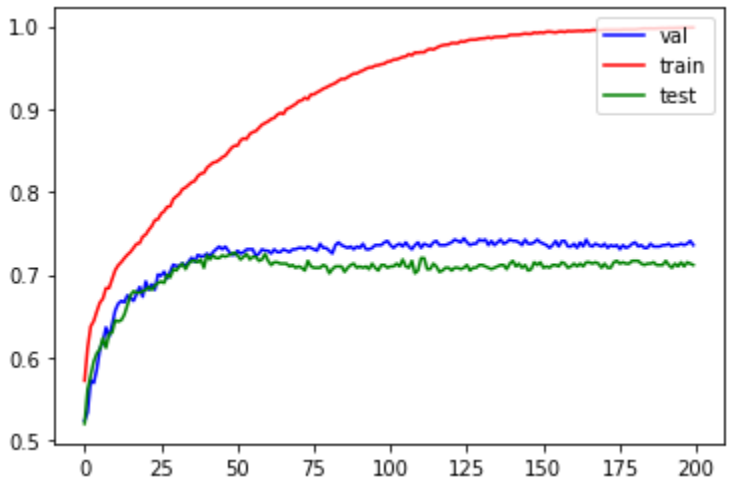
These metrics are used together very often. Precision is a percentage of tweets that were marked as positive which were correctly classified. Recall is a percentage of actual positive tweets that were correctly classified.

F1 score = 2 * (precision * recall) / (precision + recall)

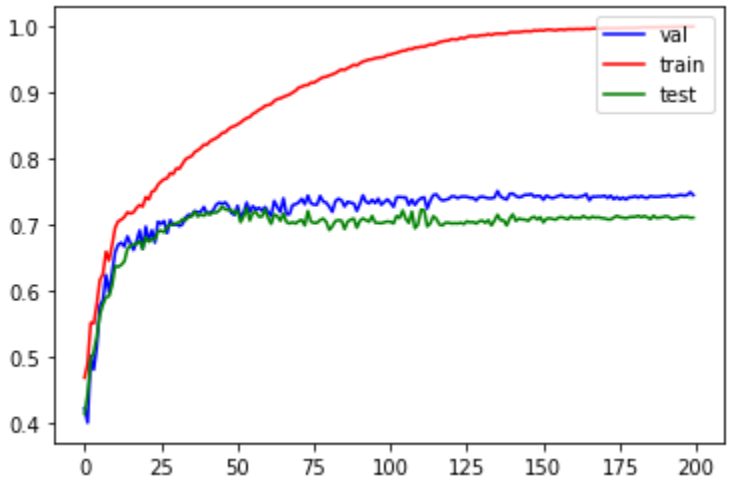
It is a harmonic mean of precision and recall.

SPLIT 1

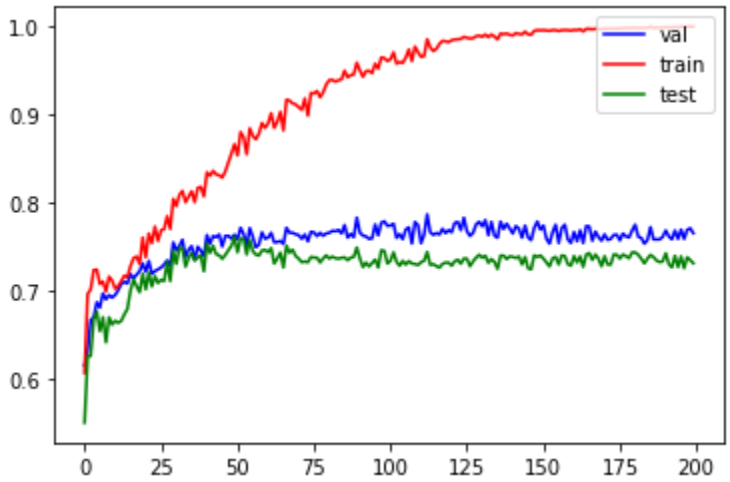
accuracy



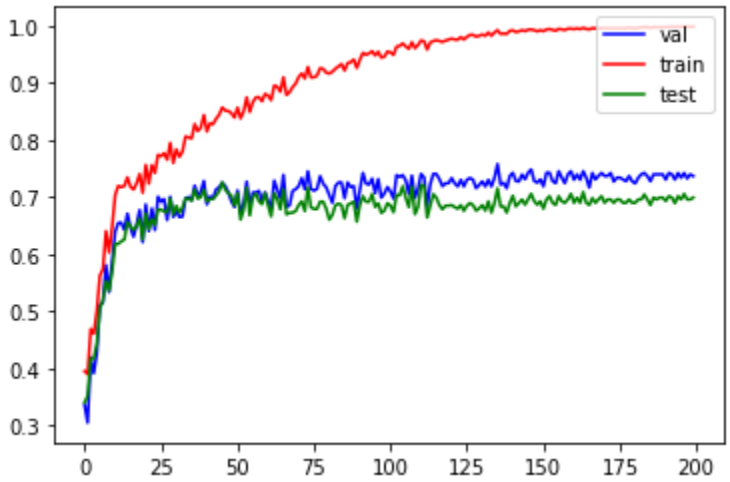
f1 score



precision

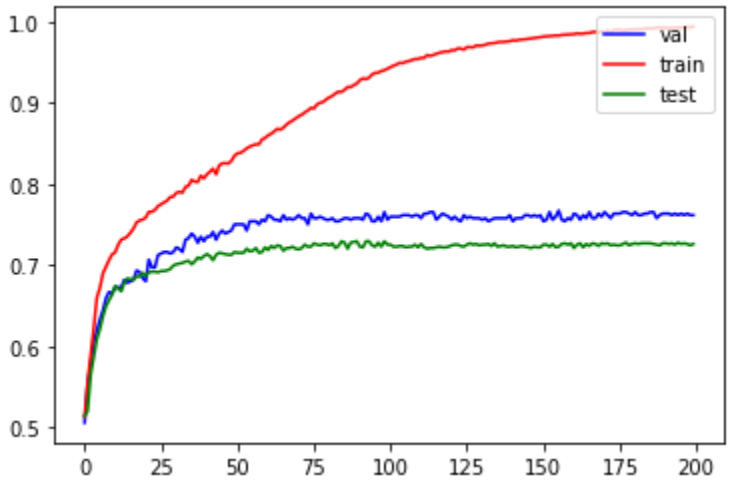


recall

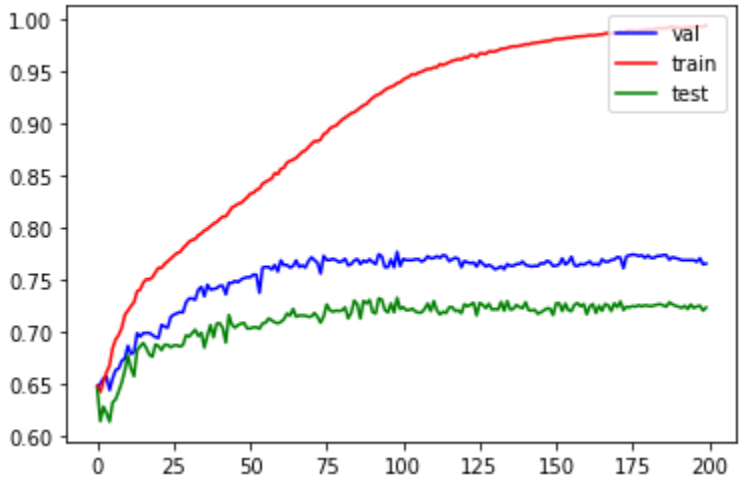


SPLIT 2

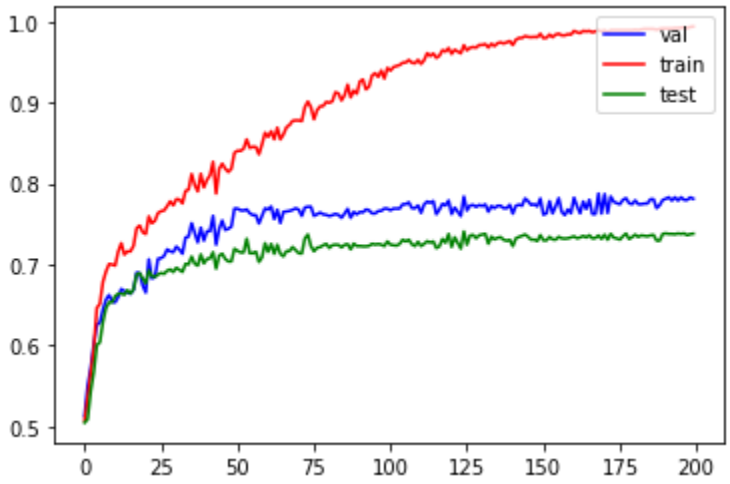
accuracy



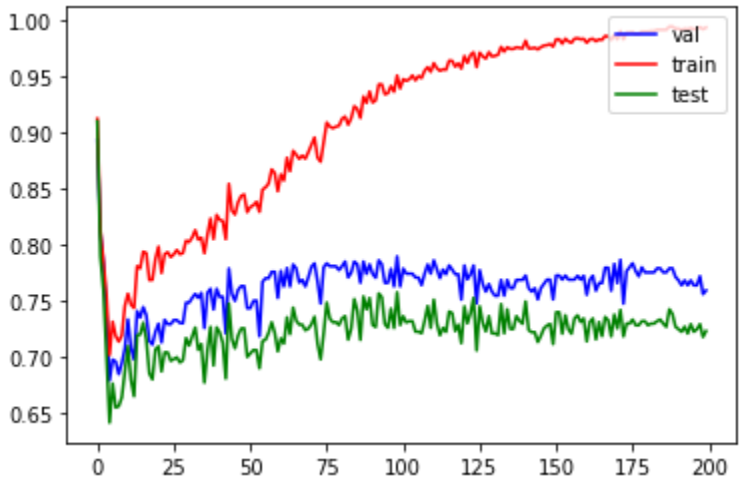
f1 score



precision

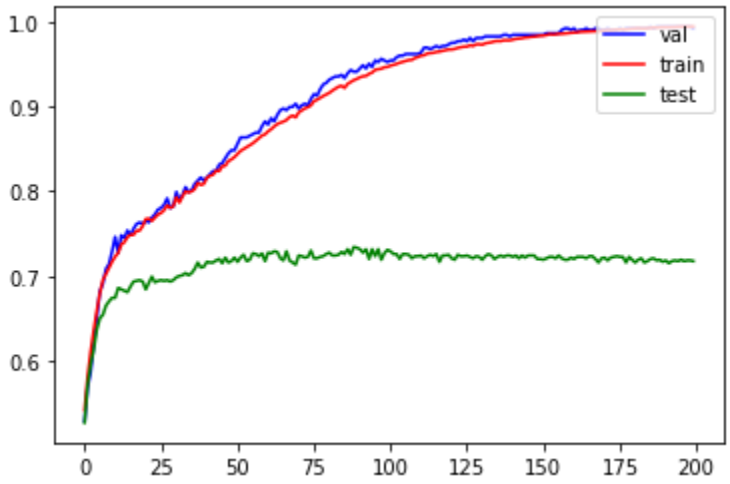


recall

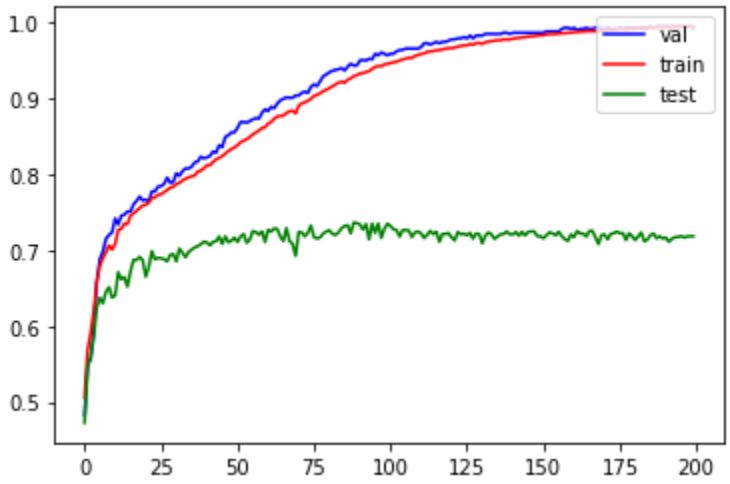


SPLIT 3

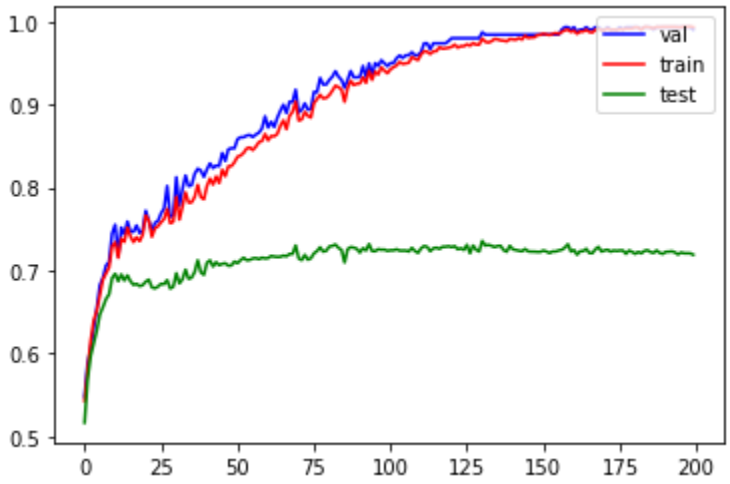
accuracy



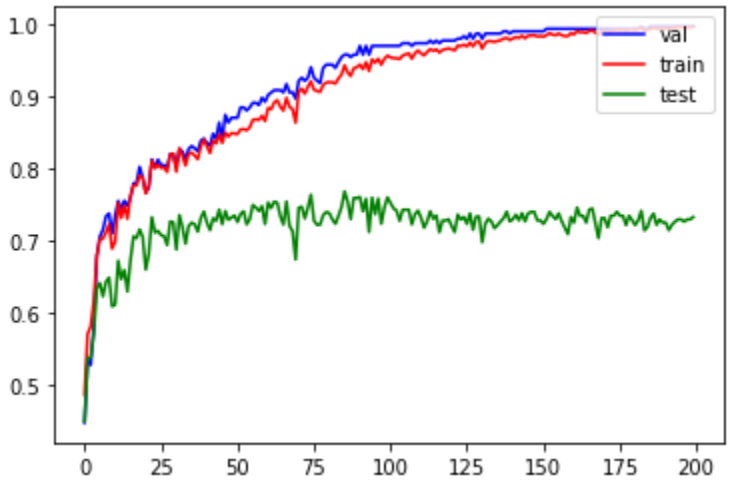
f1 score



precision



recall



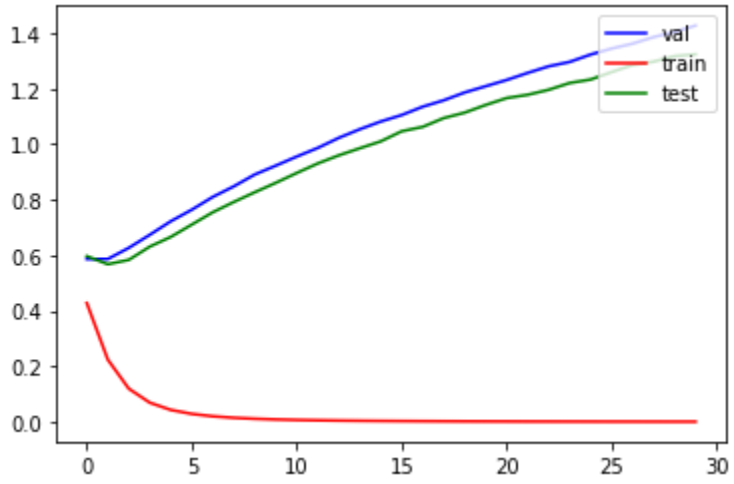
All of the splits performed very well on every metric on the train dataset. It was because models were trained on them so it was easy for it to distinguish between positive and negative tweets. Different story was for validating and testing data. For split 1 and 2 results were pretty similar. It was roughly 0.72 score for validate and 0.70 for test data in split 1. In split 2 it was slightly better (0.75 and 0.72). We can see that validating data in split 2 scores was very good. That was because in that split validating data was added to training data. Testing data in split 3 performed similar to split 2.

After all I tried to improve our model with:

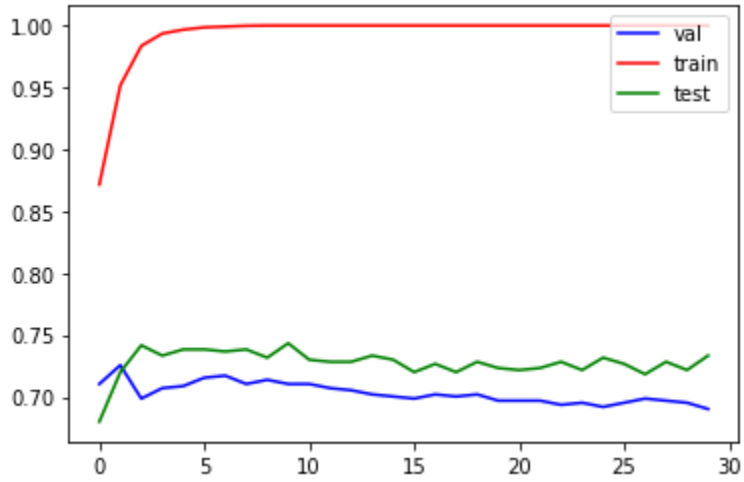
1. Changing optimization algorithm

For all training I was using the stochastic gradient descent algorithm and I decided to change it to an algorithm called “Adam”. The results:

Loss



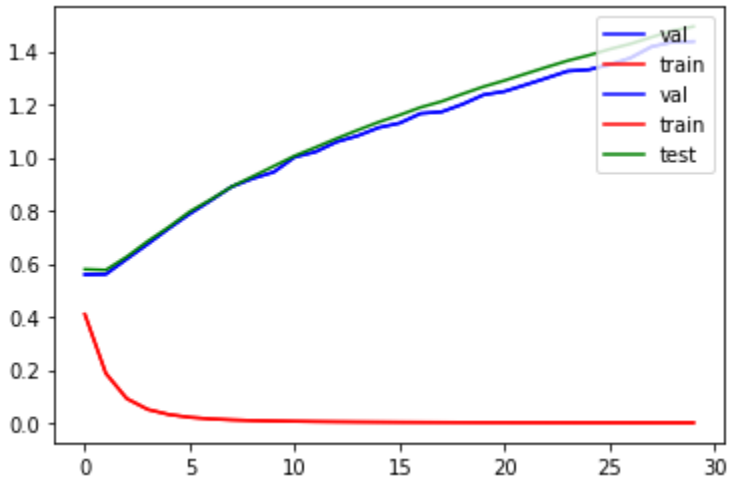
Accuracy



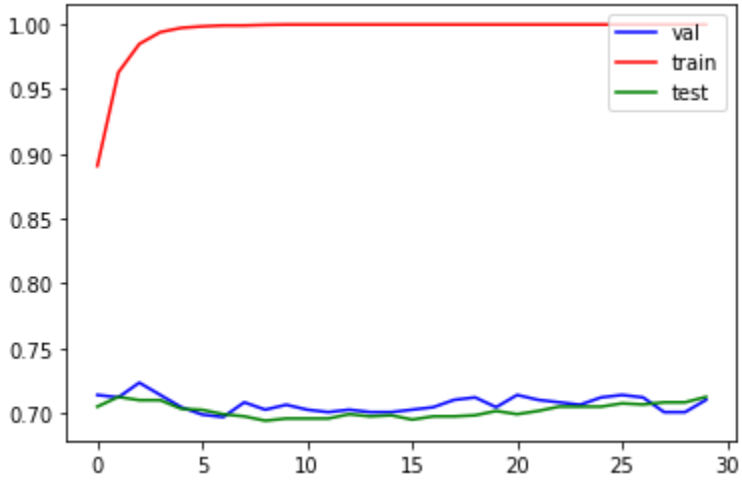
We can see that with this algorithm the model was scoring similar results but learning much faster with 5-10 epochs compared to 50-75 epochs in SGD.

2. Changing number of perceptrons

Loss



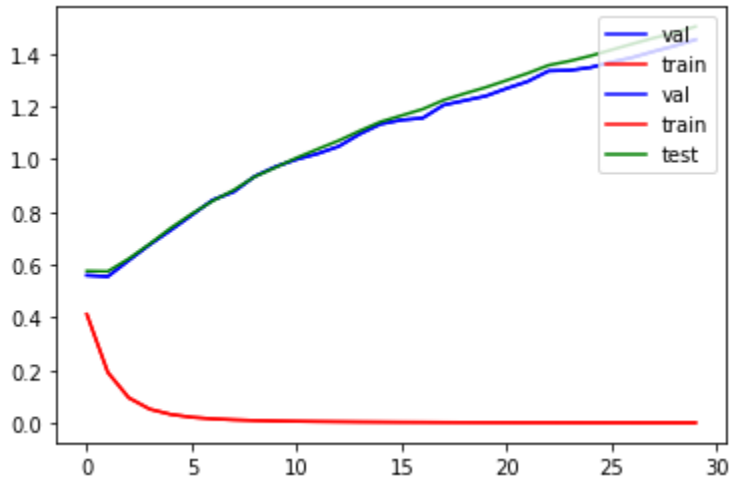
Accuracy



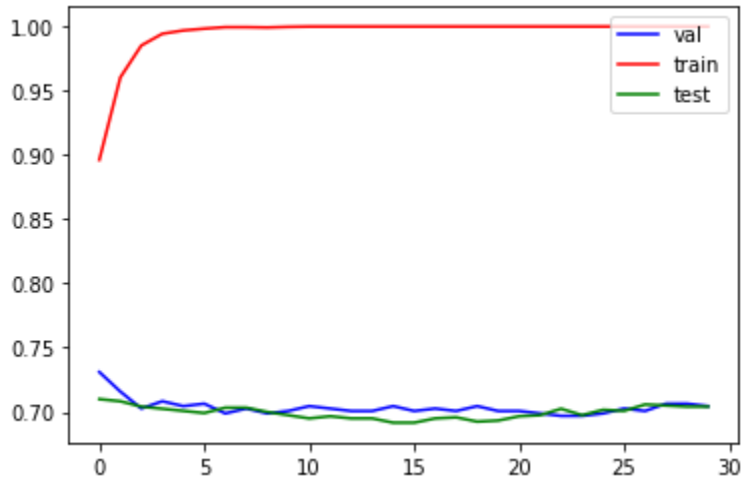
Before I was using 50 perceptrons and I changed it to 100. As well we can see the speed up in learning from 5-10 epochs to 2-4 epochs.

3. Changing learning rate

Loss



Accuracy

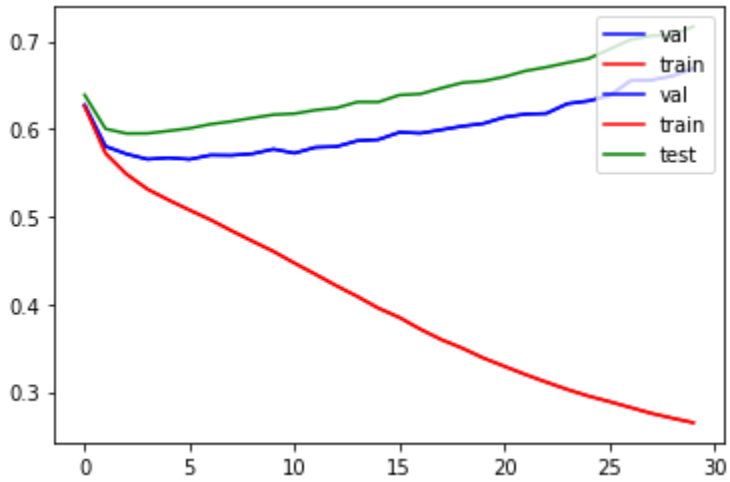


It was even quicker but still no improvements to the accuracy.

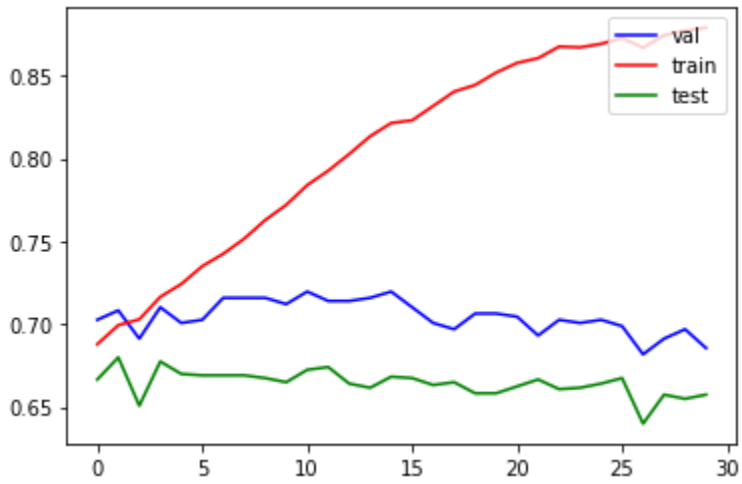
4. Shrinking the size of the vectors

SIZE = 200

Loss



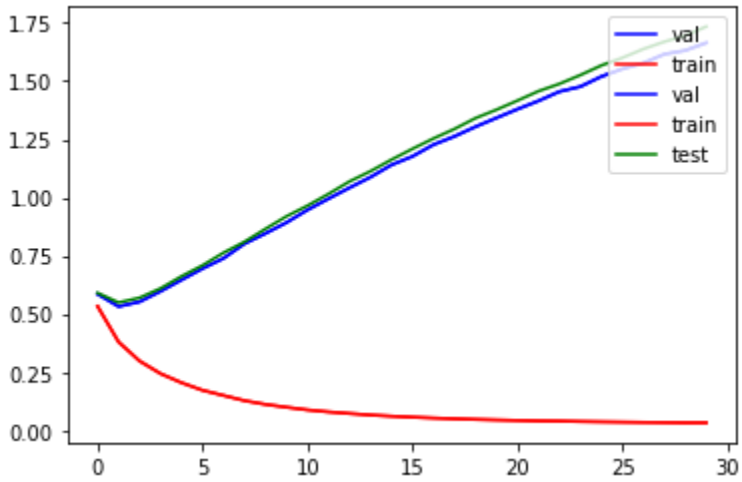
Accuracy



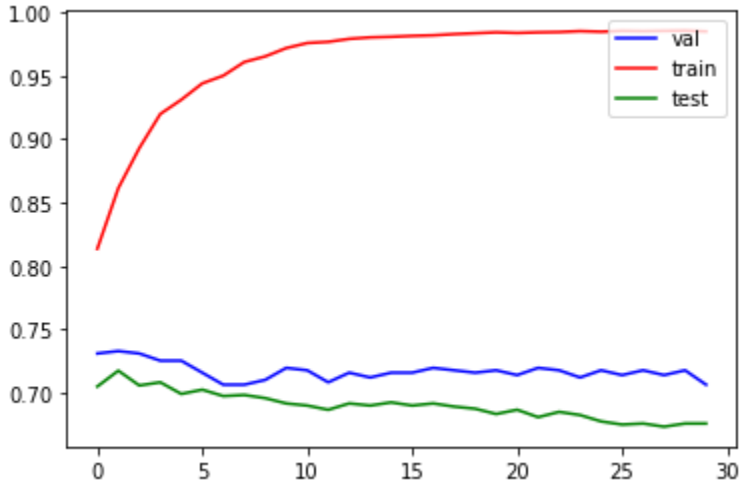
We can see that accuracy decreased so it wasn't a good move.

SIZE = 2000

Loss



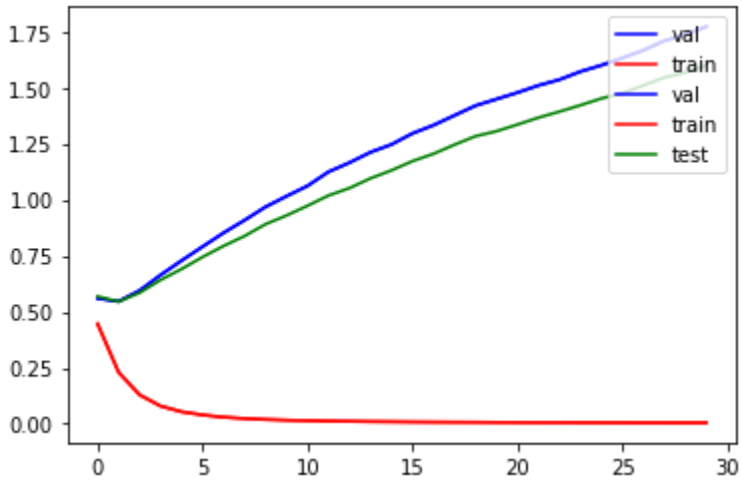
Accuracy



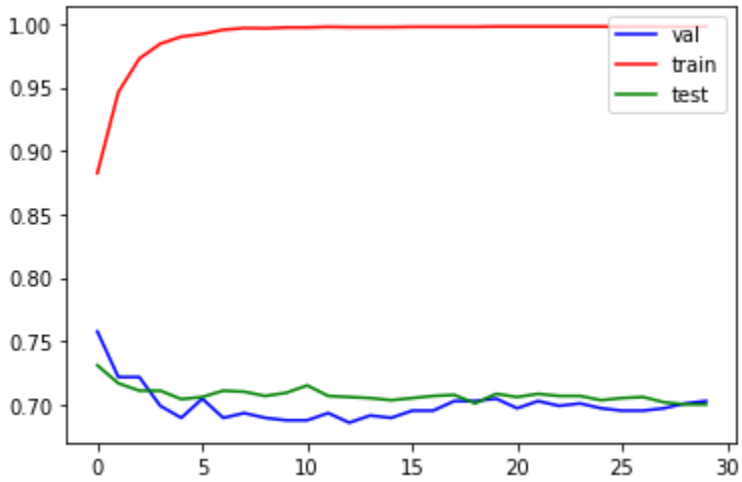
Now it was very similar to learning with initial vector size (14000).

SIZE = 10000

Loss



Accuracy



Looking at all results I think that the best model would be with vectors of size between 2000 and 10000.