Deep learning workshop - assignment 3

Doron Laadan.

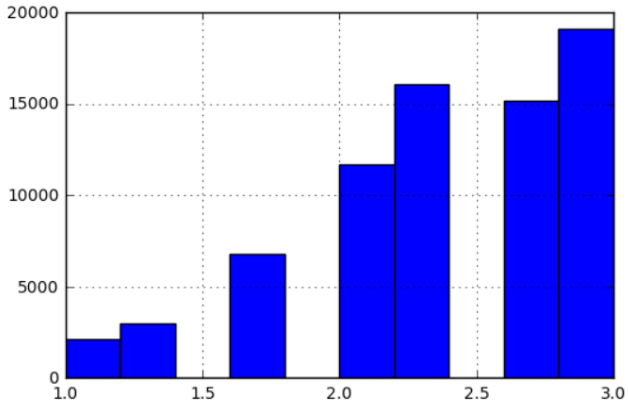
Tal ben-senior.

Preface:

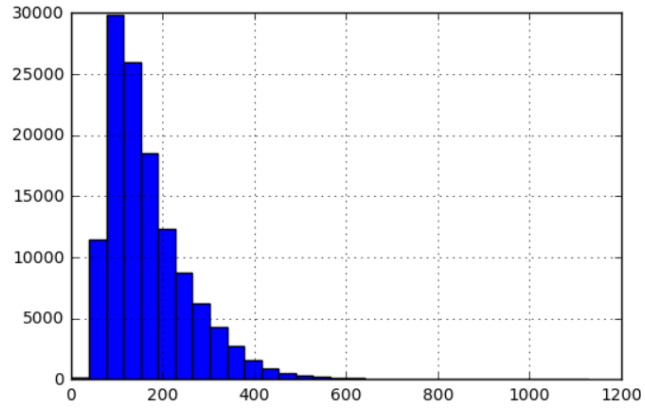
In this assignment we worked on the Home Depot dataset which was part of a Kaggle competition 3 years ago to help improve customers shopping experience by developing a model that can accurately predict the relevance of search results. Our goal is to state the relevance of a search phrase to a corresponding item. We used this dataset and assignment to learn a new ANN architecture calls "Siamese networks" that is useful when comparing inputs from similar domains and practice word embedding in different level.

We didn't perform EDA on the data ourselves but relied on EDA kernels posted on Kaggle in the competition page. An example of the analysis we used is shown here:

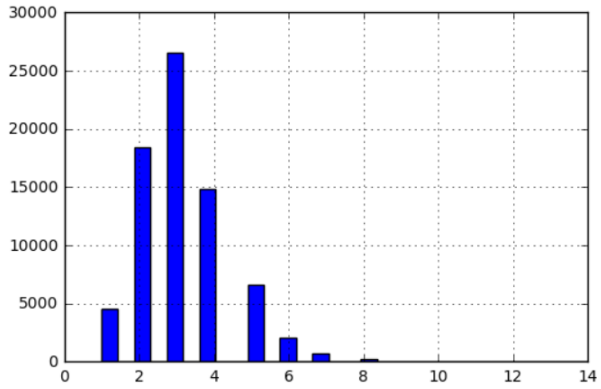
Relevance score distribution



Product description length distribution



Search terms length distribution



To tackle the problem we used several methods, first we created a naïve baseline for the problem. second, we used character level embedding with a LSTM model in the Siamese architecture and novel machine learning models with features extracted from the network. lastly, we used word level embedding with a LSTM model in the Siamese architecture and a novel machine learning models with features extracted from the network. In the following section we will describe each of the methods and the work flow of the assignment, while visualizing the result and process.

General pre-process:

for the assignment at hand we performed minimal preprocessing on the home depot data.

The only thing we did, which we did in every section, was to merge the train and test data frames with the product description data frame so we can access the product description, name, search term used and relevance score in one dataset.

For the test evaluation, instead to submit each time our result in Kaggle, we used Kaggle published results, contains the public and private records. Some of the record were with relevance value of '-1', thus we decided to ignore them. We merged the test set same as before to get the relevant features we are using later.

Part 1: naïve base line:

Main idea: count the number of times each word in the search term appeared in the product description and using that simple feature for a linear regression model.

Pre-process:

We used porterStemmer to stem the "search_term" and "product_description" columns on the train and test data frames. We used a popular implementation for the stemmer from the "nltk" package. The idea beyond using stemming before the word count is a basic process in text processing for such tasks and we used similar method last year for an Information Retrieval project. Stemming is the process of reducing inflected words to their word stem, base or root form, generally a written word form. It is usually enough that related words map to the same stem, even if this stem is not in itself a valid root. In that way the words count can be better because its work on the stem of the words.

Feature engineering:

We added a new feature which was a simple count of the number of words in the search term that appeared in the relevant product description.

Before using stemming we got on average 5 word count for each search but after the stemming process those number went up to around 8. So it's wasn't that much of an improvement but we could see it let us 'catch' more words between the search term and product description which we think helped the model (the validation score was a bit higher too).

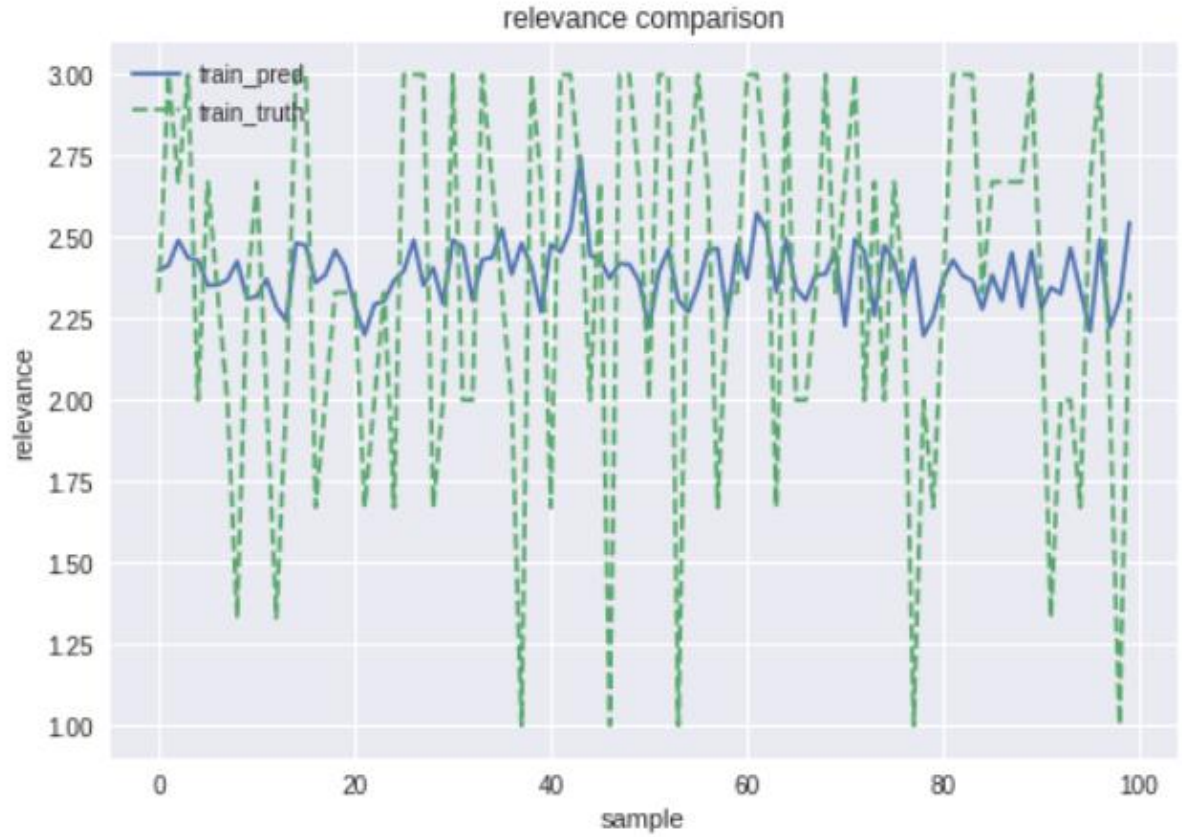
Model and results:

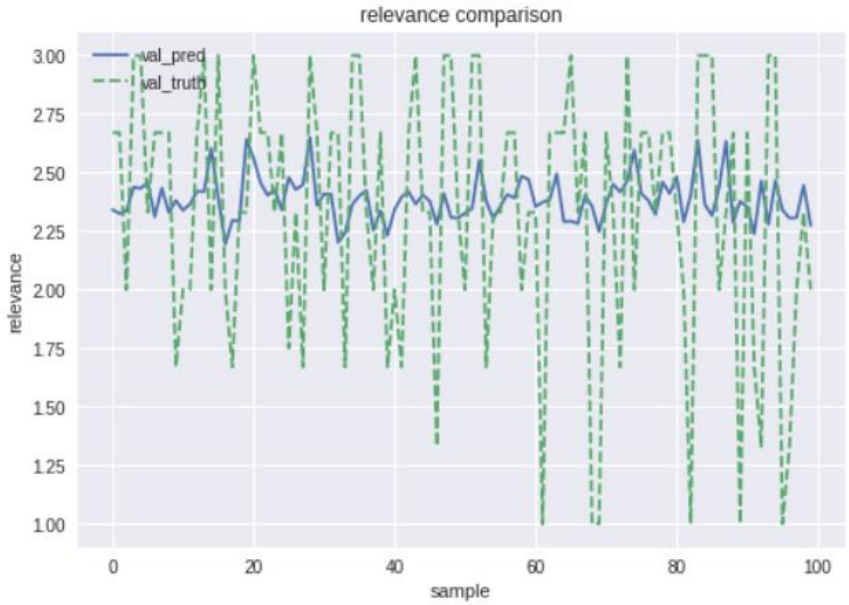
The model that we used was linear regression model from sklearn. We used a train test split with the default values of the sklearn function which resulted in a 80-20 split to the train and validation respectively.

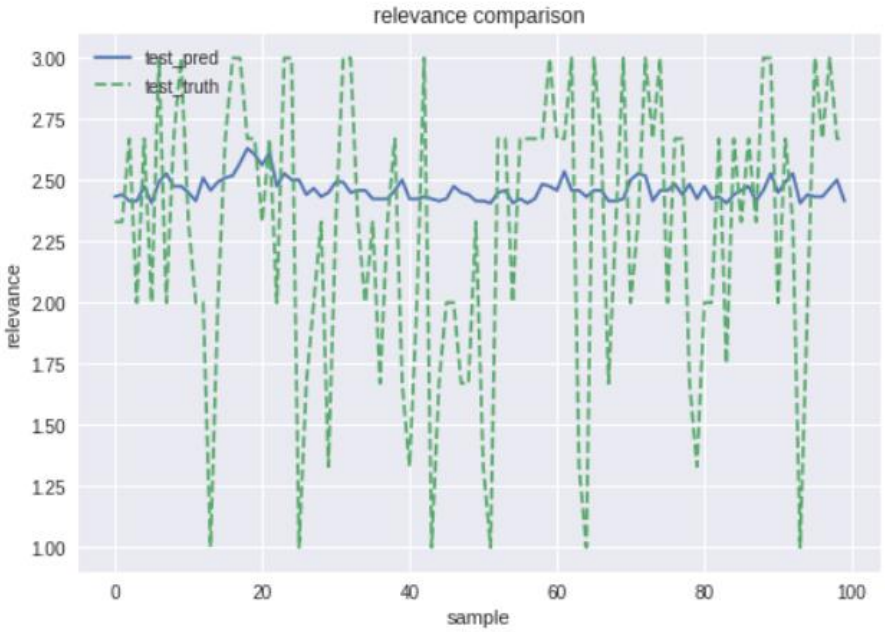
We than fitted the model on the train and predict on the train, val and test dataframes. The results can be seen here (after using stemming):

```
Running time: 0.0067310333251953125
---------------------------------
Train RMSE loss: 0.5268243444097815
Train MAE loss: 0.4326498160610803
---------------------------------
Validation RMSE loss: 0.5239085804640511
Validation MAE loss: 0.43064882315047864
---------------------------------
Test RMSE loss: 0.5272216865435995
Test MAE loss: 0.4329597361567749
---------------------------------
```

Let's look on the first 100 predictions vs the real values in each of the cases.







As we can see the train validation and test set gets pretty much the same results on the RMSE and MAE metrics we used, you can see that the RMSE score is higher than the MAE score. This might be because of the nature of the metrics, while RMSE punished big differences between the prediction and the real values. We can also see from the graphs that the model learned to "stay safe" most of the prediction he made were around 2.5 relevance without any 'bold steps', so we can assume the naïve model simply learned the mean relevance value and used it.

Part 1: character level:

First, we computed the embedding representation of the characters, as we thought the assignment is focused on the implementation of the embedding itself, we implemented it with simple methods inspired from word2vec.

The network we used for the embedding contains the input which is a single character, and the output which is a tuple of the previous and next character. The words we choose had been taken from the 'search term' column.

For the word 'Monk' the training samples are:

Input	**Output**
M	(_, o)
o	(M, n)
n	(o, k)
k	(n, k)

The network architecture we used is as follows:

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         (None, 1)                 0
_________________________________________________________________________________________________
char_embedding (Embedding)   (None, 1, 16)             1456        input_1[0][0]
_________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 16)                0           char_embedding[0][0]
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 91)                1547        flatten_1[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)              (None, 91)                1547        flatten_1[0][0]
=================================================================================================
Total params: 4,550
Trainable params: 4,550
Non-trainable params: 0
```

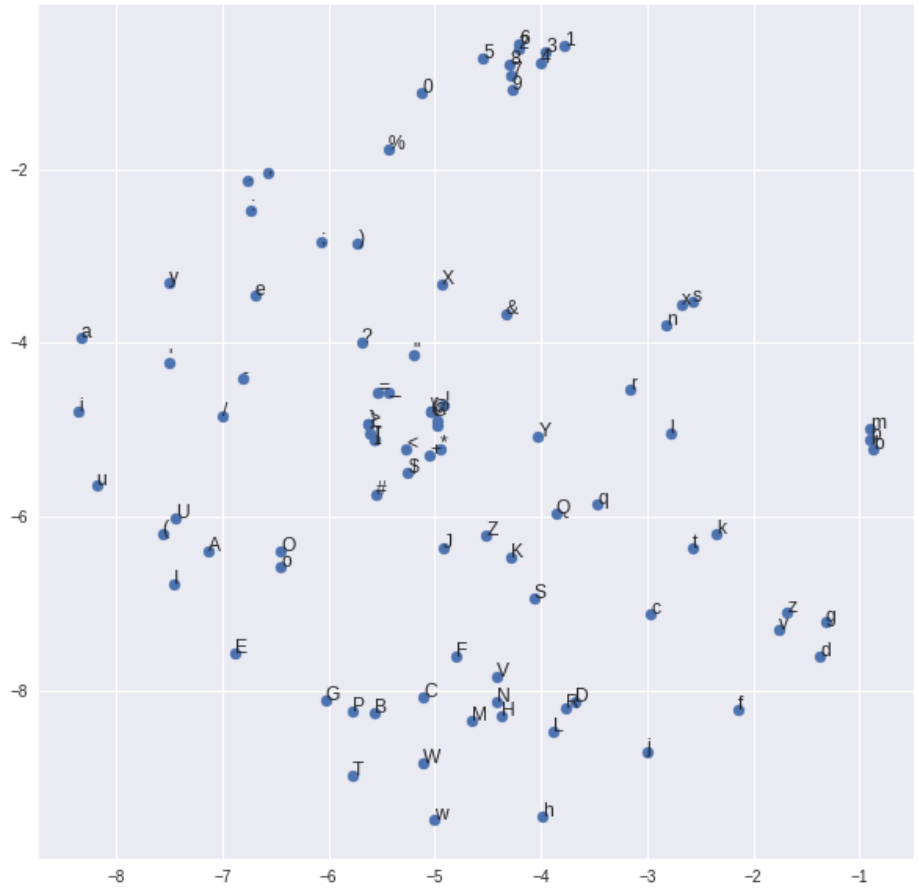
Each character is represented as 16 vector size of features. The output size is 91, which is the size of the vocabulary of the characters we created using simple parsing methods, we trained the network according to categorical cross-entropy loss where each category is character, for 3 epochs with l2 regularization at the embedding layer with parameter of 1e-3.

The reason for the l2 norm is because we want the embedding weights to be uniformly distributed and avoid situation where one feature is too dominant relative to the rest.

We also noticed that too much epochs results with unclear relations in the TSNE representation thus we decided to decrease the number of epochs to 3.

The accuracy and the loss of the model were high, yet those result are irrelevant though our goal is to find the characters context in the embedding layer, and not succeed on the network task (which makes no sense).

The results can be seen in the plot below (TSNE):

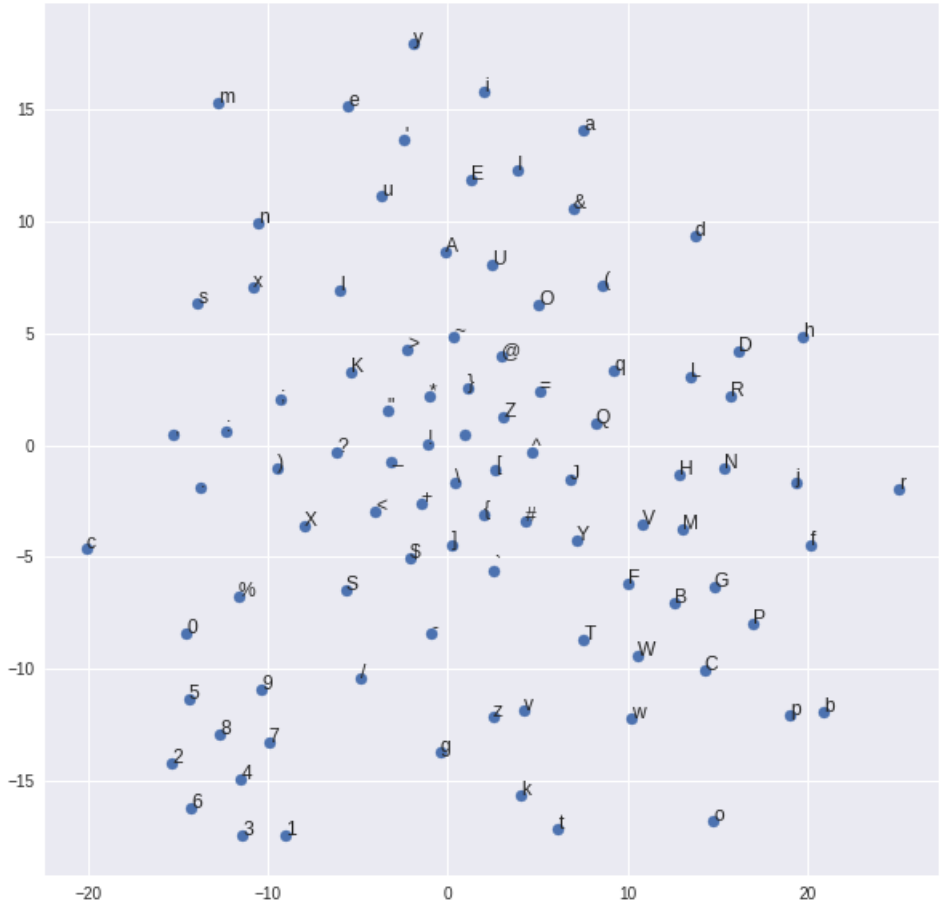


As we can see there is separation of numbers / lower case characters / high case characters. Another interesting result is the similarity between vowels.

Relying on the character embedding results we got, we chose not to implement more complex word2vec methodology such as skip grams and negative sampling.

We examined how the character representation might be different if the embedding is trained from the column 'product description' data instead in order to test if there any reason to use both embedding as inputs to the network. Because the result seemed like those we got by the 'search term' data we conclude that we can use the same embedding for both inputs of the Siamese model.

The final embedding we used trained on both the 'search term' and 'product description' data. Though it's using just the 100k first rows of each for time computing efficiency.



Siamese networks:

The Siamese network goal is to predict relevance of a search term to the product description. The input of the model is the 'search term' to the relevant 'product description' respectively, where the sequence length we chose is 30 characters each. We are aware that the average size of the product description characters size is much higher from the sequence length we chose, yet we decided to remain with this size for two main reasons:

1. Simplicity, the running time is much faster that way, and that allows us to optimize the network more easily, furthermore we consulted with other students who chose sequence length of the average/max length of the samples and were not convinced by the result they achieved.
2. We believe that the network will loss the context with too much data to process and requires more complex network architectures for such a task.

Our network final architecture is as follow:

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	(None, 30)	0	
input_5 (InputLayer)	(None, 30)	0	
sequential_2 (Sequential)	(None, 200)	175056	input_4[0][0] input_5[0][0]
cosine (Dot)	(None, 1)	0	sequential_2[1][0] sequential_2[2][0]
lambda_2 (Lambda)	(None, 1)	0	cosine[0][0]

Total params: 175,056
Trainable params: 173,600
Non-trainable params: 1,456

Where sequential_2 is the shared model with the architecture:

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 20, 16)	1456
LSTM (LSTM)	(None, 200)	173600

Total params: 175,056
Trainable params: 173,600
Non-trainable params: 1,456

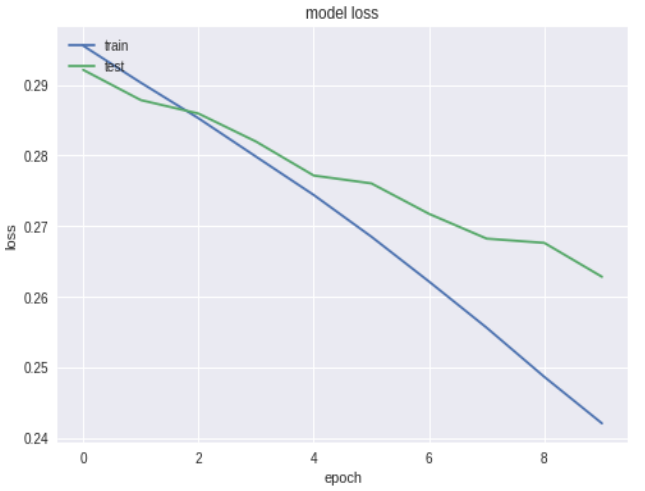
The output of the 'sequential_2' model is a 200 size feature vector for each of the model inputs, we then calculate the cosine similarity between those 2 vectors. Because cosine output is in [0,1] range we later added a custom lambda layer that calculates: $2 * (cosine(in) + 1)$, to fit the results for our data output range (1-3).

We are aware that adding dense layer instead of the lambda layer may be used as a normalization as well, but the way we saw it is, the two vectors the Siamese model outputs before the cosine layer should represent the distance between the two layers, whereas the dense layer might overfit our data rather than find the layers similarity.

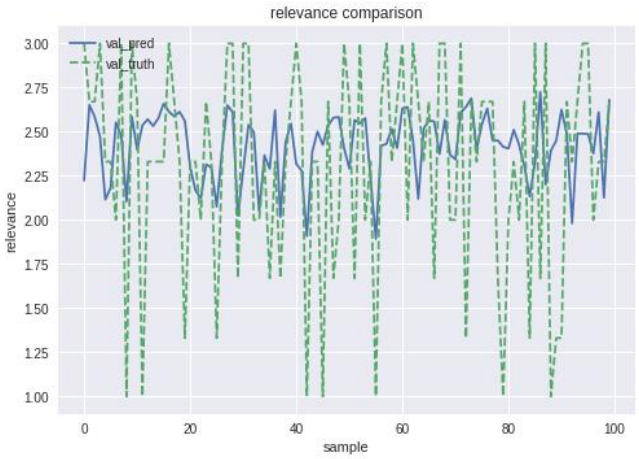
We trained the network with batch size 64 for 10 epochs, with 0.8 of the training data used for training and 0.2 for validation.

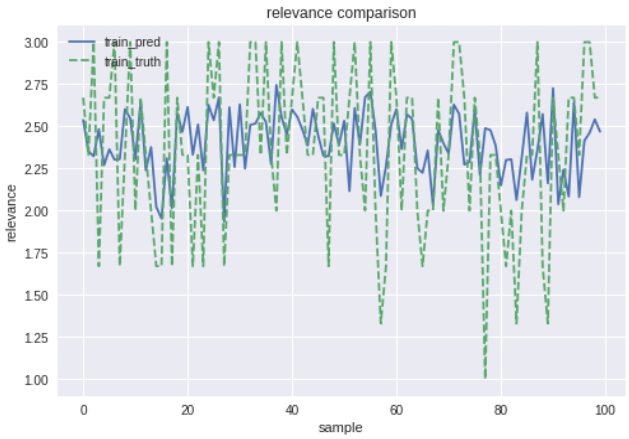
The results we got:

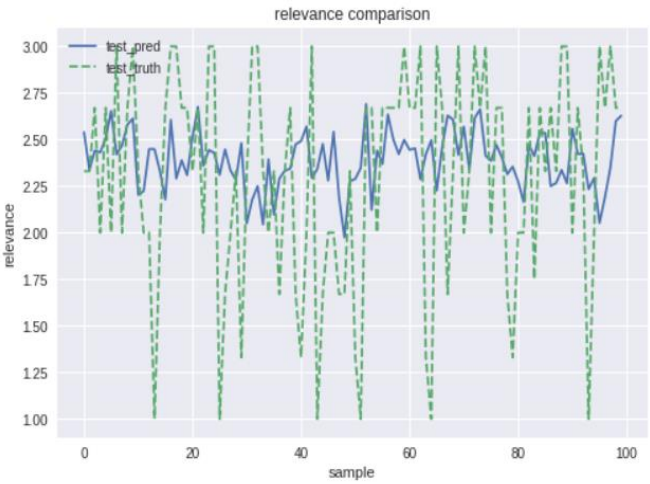
We can see the trainng and test loss for each epoch the final results and runtime and the predictions values against the real values in the the different data set(train,val,test).




```
Running time: 1151.7979772090912s
---------------------------------
Train RMSE loss: 0.4903181826915169
Train MAE loss: 0.40011738754329945
---------------------------------
Validation RMSE loss: 0.5225450774302781
Validation MAE loss: 0.4263533351771243
---------------------------------
Test RMSE loss: 0.5369353088910119
Test MAE loss: 0.43751141981023456
---------------------------------
```







As we can see our model has similar results on the validation set as the naïve model we used. We can also see that the model not tend to go too much outside the 2.5 relevance score even when the real relevance score is much farther, yet the model prediction seems bolder.

Later we tried to overfit our training by increasing the number of training epochs, the results are not documented but we could see the model is much 'bolder' than before. Unfortunately (although it could be expected) the validation loss was much higher, which explains the willingness of the model not to predict too further from 2.5 relevance in the previous models.

Custom loss optimization:

The model loss usually used for such tasks is MSE, which computed:

$$MSE = \sum_{i=1}^{i=n} \frac{(P_i - O_i)^2}{n}$$

Where P_i is the predicted value of the network, O_i is the goal truth value, and n is the batch size. One can note that in cases where the differences of the predicted value and the goal truth is higher than 1, the loss increases, else the loss decreases (difference smaller than 1), which may cause the model to avoid predicting values further from the average values of the goal truth.

This thought led us to build custom loss which fits the task better (or at least we hope will fits better).

Instead of using MSE loss only, we use MAE as well, which his formula is:

$$MAE = \frac{1}{n} \sum_{i=1}^{i=n} |P_i - Oi|$$

It's seems like the MAE loss is more fit for tasks where the difference of the predictions and goal truth values are less than 1 (by punishing those prediction more than MSE does).

The loss function we defined works as follows:

Keras loss function get batch of samples (not a single sample)

Function compute_loss(y_pred, y_true)

mse_samples → samples where y_pred – y_true > 1

ase_samples → rest of the batch samples

mse → mse_samples error relative to mse_samples length

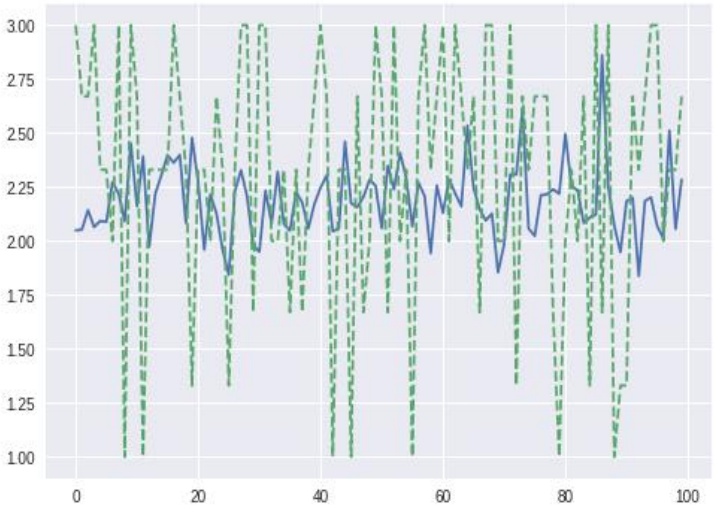
ase → ase_samples error relative to ase_samples length

return ase + mse

Results:

```
Running time: 937.2493093013763s
---------------------------------
Train MSE loss: 0.5577856127314151
Train MAE loss: 0.47334341603343605
---------------------------------
Validation MSE loss: 0.5678256500211512
Validation MAE loss: 0.47723319638285605
---------------------------------
Test MSE loss: 0.5790501037695207
Test MAE loss: 0.48790674084499797
---------------------------------
```

Train:



At first look, the result not impressive at all, yet it's seems we succeeded in achieving somewhat satisfies result – we can see in the graph that the model prediction is bolder than before (though it just 100 samples), in the 90-sampling area the model prediction was almost 3 (though his prediction was wrong).

Some points we thought about while doing the assignment but were not investigated due to time constraints:

- Using just the MAE as a loss function and compare the results (MSE/ASE/ MSE+ASE), we may conclude that the result is like those we achieved - which means our custom loss function is bad in every aspect.
- Checking the number of peaks (prediction higher than 2.75 or lower than 1.25) and compare the RMSE of those peaks relative to all the models with the different losses.

Those suggestion may not decrease the loss of our task regarding the RMSE metric, yet we think further investigation of the effects of the loss functions to the model predictions could be quite interesting.

For the rest of the report, we are using the same architecture we used before the custom loss model.

Feature Extractor:

Next (as we learn to love in this course), we used the ANN network built in the previous section as a feature extractor for a classic machine learning algorithm. We used the output of the LSTM layer in the shared model which resulted in 400 features for the ML model (200 for each input). We choose 2 of the state-of-the-art ensemble methods to test, the first is XGBRegressor which is an ensemble model based on iterative process and boosting and the second is Random Forest Regressor which is an ensemble model based on non-iterative methods and bagging. Both models show good result and usually outperform other methods, so we decide these two.

XGB-Boost:

At first, we got into a lot of pain with this part. We used the features we extracted with default hyper parameters values and the model couldn't learn (Spoiler). This resulted in validation score of above 3.1 (MSE). So, at that point we assumed something has gone wrong but didn't know what it was.
Was it the features we extracted or maybe the model isn't good enough? The second option was unlikely, we already achieve reasonable result in the previous models.

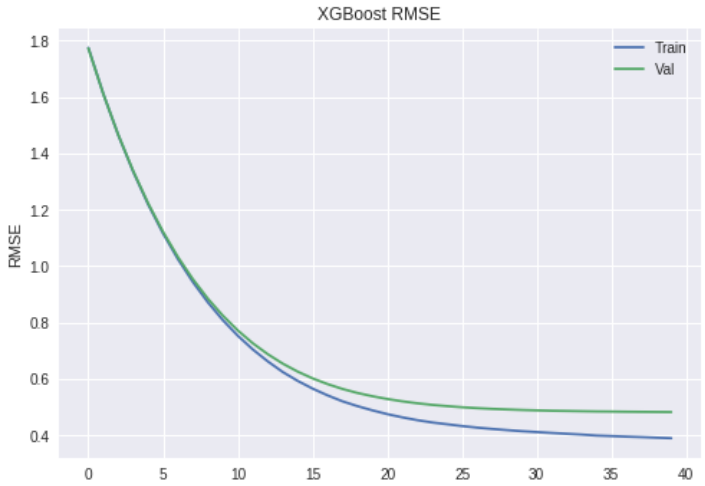
First thing we did was restarting our environment and rerunning everything – that (of course) did not work.

Next, we tested our features – we loaded the model and extracted the features as mention before but instead of using them as input to a ML model we wrote a cosine function same as we implemented in our model and normalized the cosine output. Unfortunately, the result was correct – like the one we got from the original model. Well, the feature extraction worked fine. .

At this point we felt pretty discourage and thought that maybe classic ML aren't any good here, maybe there were too much features (400) for the ML to learn the similarity function, we also tried to use PCA to reduce features dimension with no success. After we consulted with other students who did got reasonable results we tried again. This time we ran grid search on the model hyper parameter (not in the colab due to time constrains) and SHAZHM! The model with the tuned hyper parameters was able to learn!

We ran the model with n_estimators = 40, learning_rate = 0.1, max_depth = 10 and results are as follows:

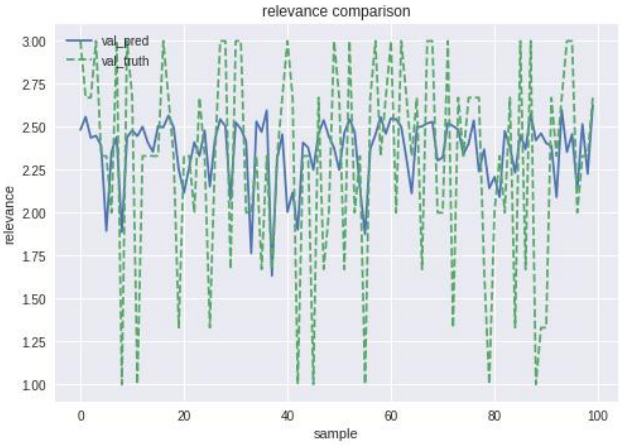
We can see the learning rate, for XGBRegressor the x-axis is the number of trees in the ensemble each iteration in the boosting process. And we can see improvement in the score on the validation set. Unfortunately, there is a case of overfitting we couldn't resolve (if we wanted the model to learn)

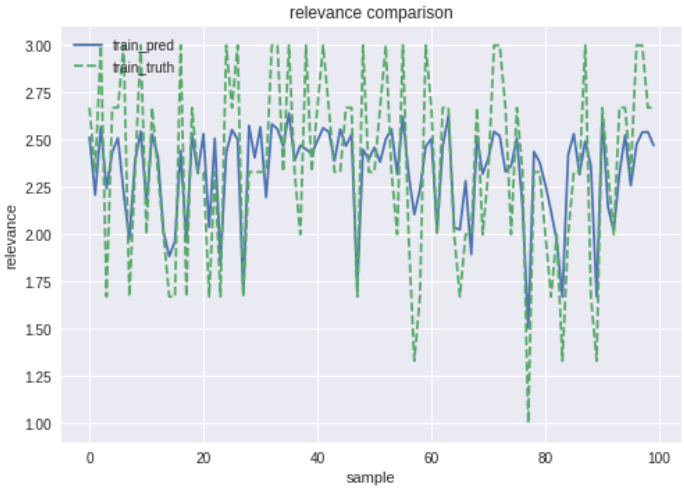


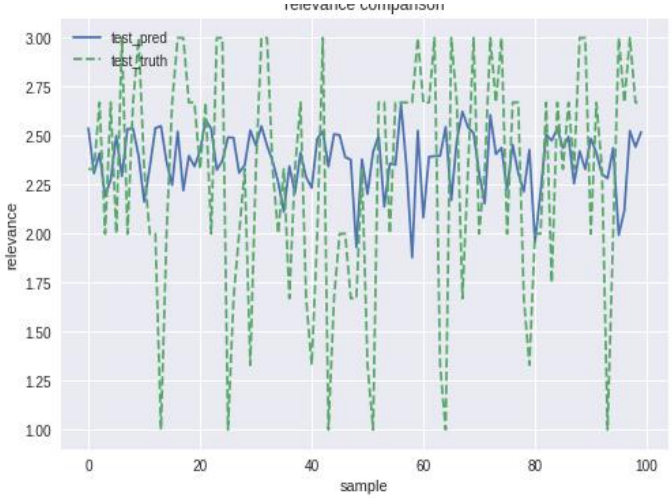

```
Running time: 235.1116805076599
---------------------------------
Train RMSE loss: 0.3894864000384748
Train MAE loss: 0.32416903709119144
---------------------------------
Validation RMSE loss: 0.48832708813593007
Validation MAE loss: 0.40267603855666095
---------------------------------
Test RMSE loss: 0.5282434740093834
Test MAE loss: 0.4346251281063211
---------------------------------
```

The predication against the real results on all the 3 data sets are shown here.

As we can see we indeed slightly overfitted our model on the train set. But got reasonable result on the validation set while for the test the model stayed in the 2.5 'comfort zone'





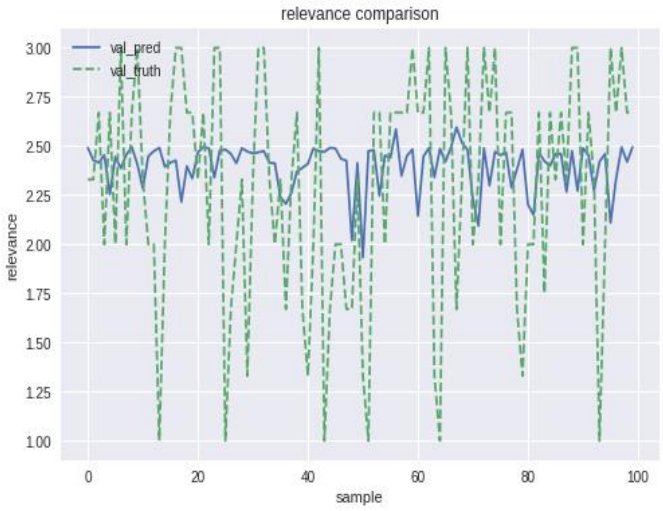


Random Forest:

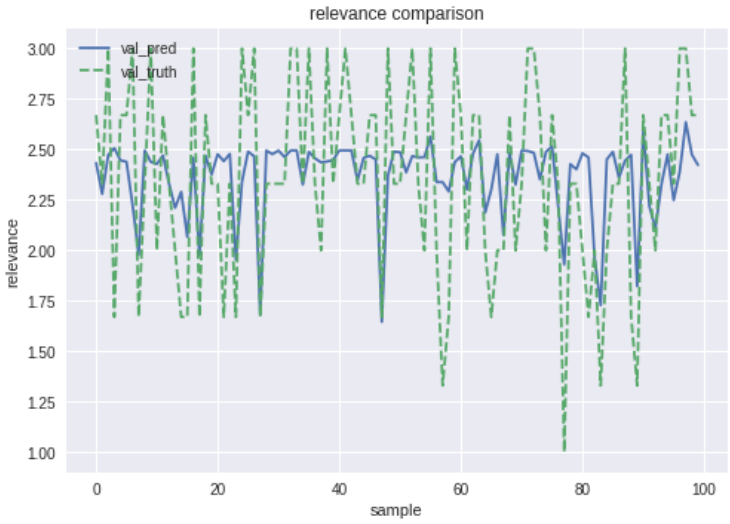
For random forest we followed the same process as before. this time no tuning was required and we just used n_estimators = 15, max_depth = 10 to produce the following results. Again we have a slight overfit but the results on the validation are better than the ANN network and naïve baseline although not as good as the XGBRegressor.

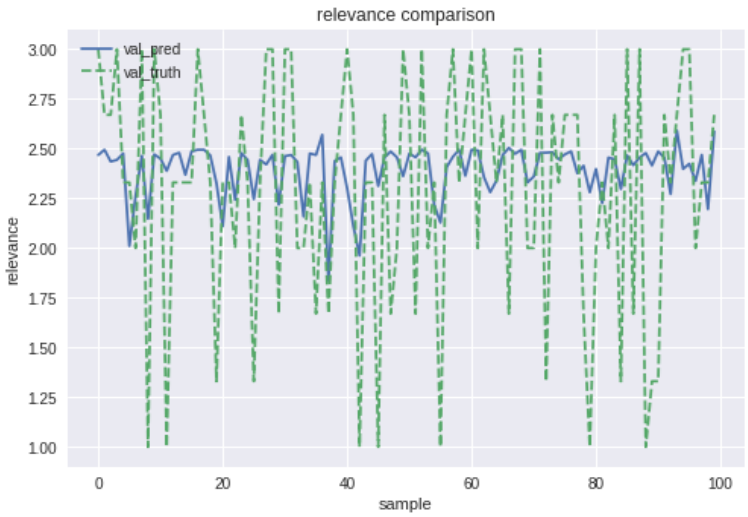
```
Running time: 269.6119918823242s
---------------------------------
Train RMSE loss: 0.4698523672898325
Train MAE loss: 0.38935987107010384
---------------------------------
Validation RMSE loss: 0.4982536654968923
Validation MAE loss: 0.4101006978664676
---------------------------------
Test RMSE loss: 0.530534572129795
Test MAE loss: 0.4360223004401761
---------------------------------
```

test



train





We can note nice phenomenon in the graphs we plotted:

Almost all the results are below 2.5, even at the peaks at the training where we overfitted the data the model avoids predicting values higher than 2.5, though he does predict peaks with low values. We assume the reason is that most of the data 'relevance' is closer to the maximum value (3) rather to the minimum value (1), then the model punishing these low values while 'prefers' to avoid predict values closer to 3.

Part 2: word level:

In this part we used word embedding level with a LSTM model as part of the Siamese network. The network architecture is the same as in part 1, the difference is the input is now the word embeddings.

Pre-process:

In contrast to the first part we took the easy path and used a pre-trained embedding model for this part. We used the well-known GloVe vector representations for words which can be read about more here: https://nlp.stanford.edu/projects/glove

Specifically, we used the Wikipedia 2014 + Gigawords vectors and the 100d vector in it so our embedding size is 100.

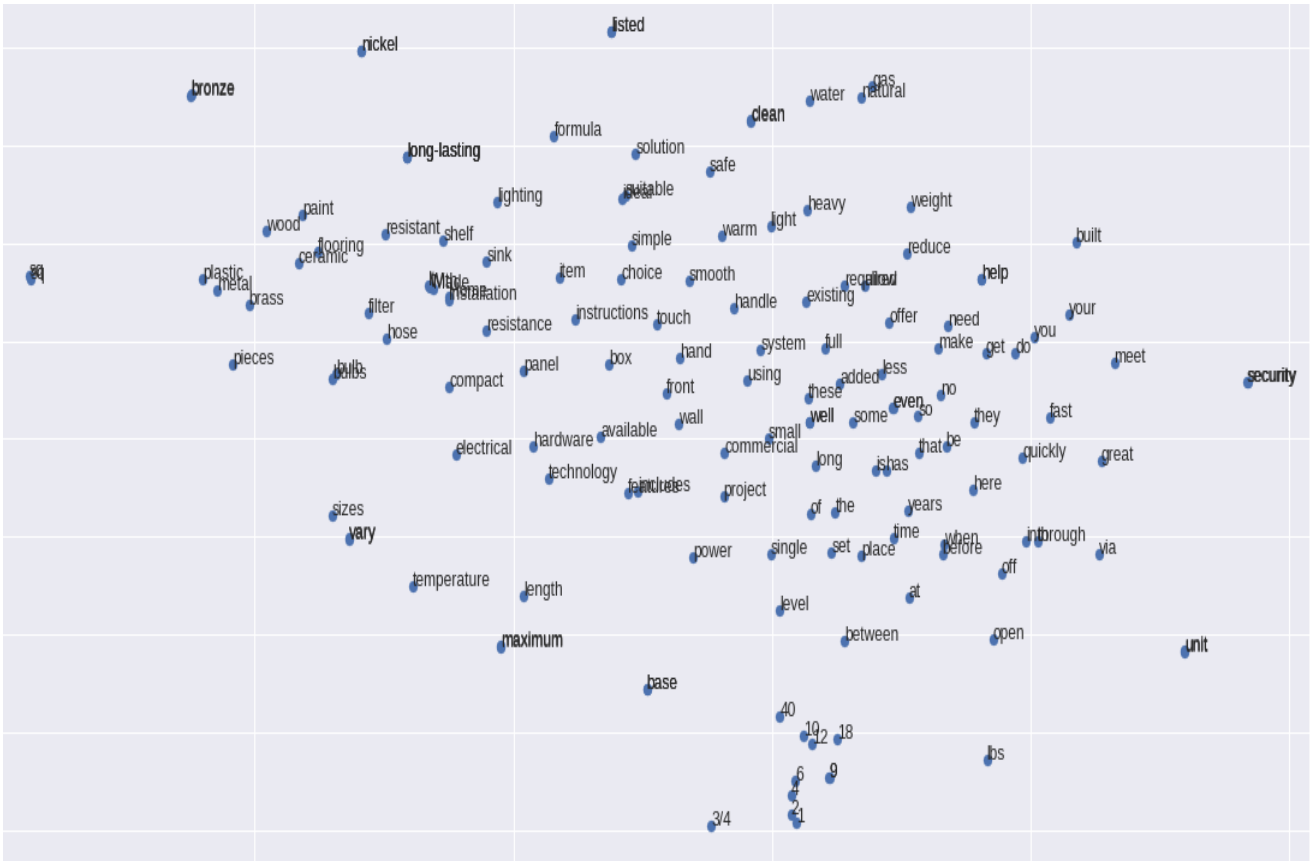
We didn't set number of unique words to use by the tokenizer because after reading blogs on the subject it appears the keras tokenize implementation doesn't really take into account that parameter (and even when at first we did use it, it didn't limit the number of unique words)

For the maximum number of words to use we found the longest search term in the data and set it to it: 60.

We used the keras preprocessing Tokenizer to tokenize the search terms and product description while using lower=False which decide Whether to convert the texts to lowercase or not, we decide against it because uppercase letters can be significant in our opinion to searches. And used the filter: filters='"&*.;?[\]^`{|}~. The idea of using this filter is so symbols like %,$,- and such can be significant in a search for a specific product. We then used the tokenizer to transform the test and train data to a sequenced and used padding to the maxlen.

We loaded the embedding vectors from the GloVe file to a dictionary which was then used with our tokenized data to create the embedding matrix and the embedding layer in the network we will use. We did a sanity check to see how many of our token appeared in GloVe and unfortunately, we are missing more than half tokens, resulting in an embedding vector of zeroes. To tackle this, we decided to run the network with the embedding layer weights trainable-parameter set to True for one epoch, to create better representation for the tokens with the current zero weights vector. Then we trained the network with the weights trainable-parameter set to False.

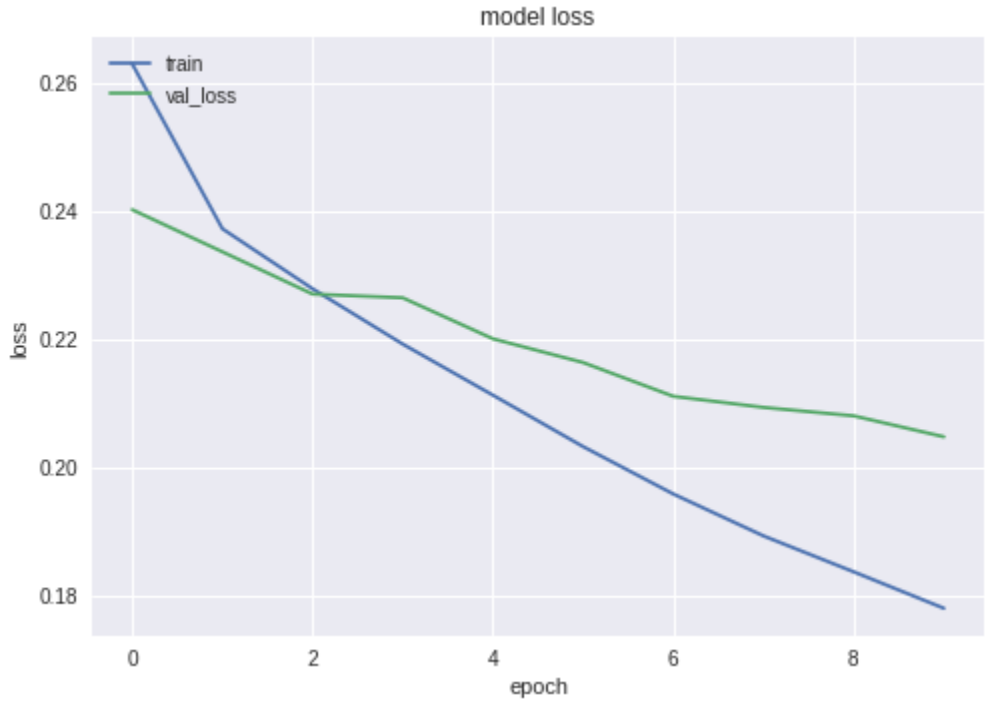
The results of the embedding can be seen here with TSNE (150 tokens out of the 500 frequent words chosen randomly):


listed
nickel
bronze
water
natural
clean
formula
solution
long-lasting
safe
lighting
heavy
weight
paint
wood
resistant
shelf
warm
light
simple
reduce
built
flooring
ceramic
sink
plastic
metal
brass
item
choice
smooth
help
handle
existing
filter
instructions
touch
offer
need
your
hose
resistance
make
get
do
you
pieces
hand
system
full
meet
panel
box
using
added
less
security
compact
front
these
no
wall
well
some
even
so
they
fast
available
small
hardware
electrical
commercial
that
be
quickly
great
long
technology
has
here
project
of
the
years
sizes
vary
time
power
single
set
place
before
through
via
off
temperature
length
level
at
between
open
unit
maximum
base
40
18
lbs
6
9
3/4

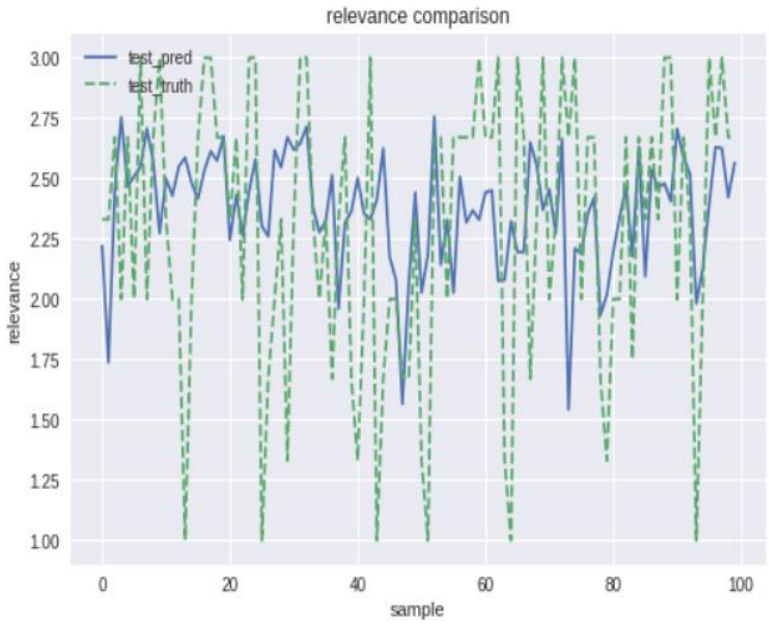
Model and results:

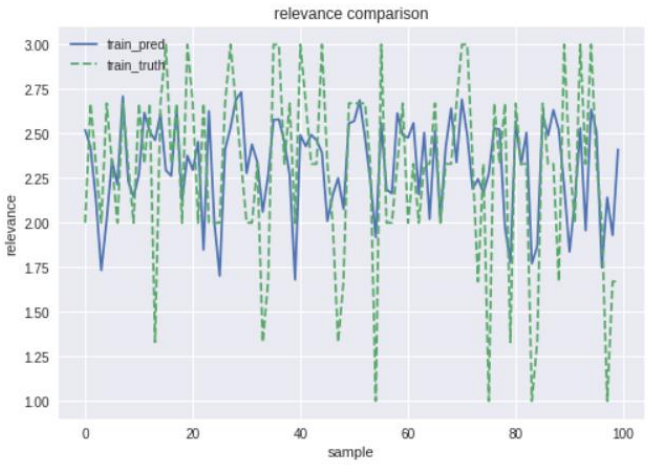
We created the Siamese network as describe in the previous part and ran for 10 epochs:

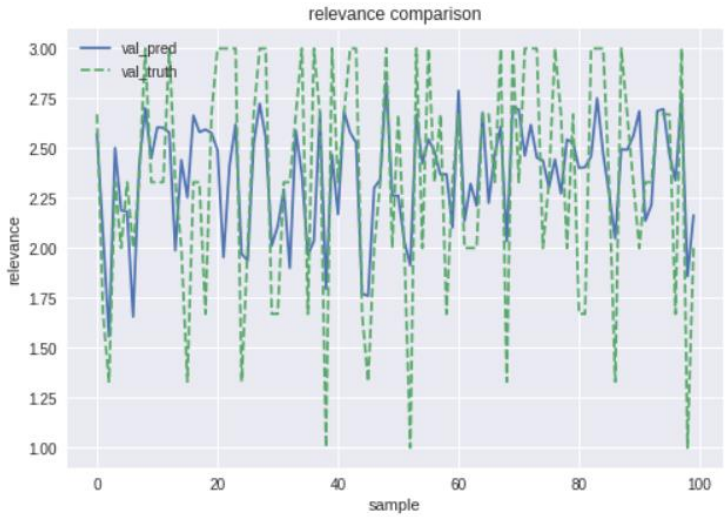
The results are as follows:




```
Running time: 1253.5856015487213s
---------------------------------
Train RMSE loss: 0.4179280623452756
Train MAE loss: 0.33792324842985066
---------------------------------
Validation RMSE loss: 0.418706552275427
Validation MAE loss: 0.33962559780724316
---------------------------------
Test RMSE loss: 0.5297903507014284
Test MAE loss: 0.4302584030588484
---------------------------------
```




relevance comparison
train_pred
train_truth
relevance
sample
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
20
40
60
80
100


relevance comparison
val_pred
val_truth
relevance
sample
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0
20
40
60
80
100

Feature extraction:

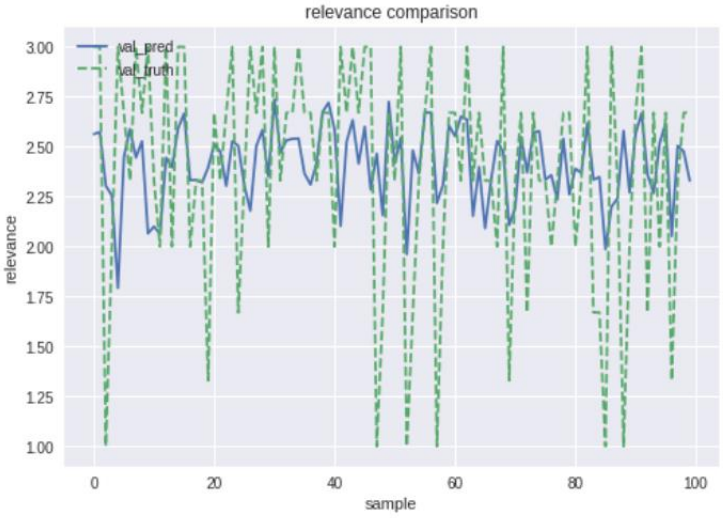
We used the network as a feature extraction for 2ML classic algorithms, random forest regressor and XGBRegressor. We choose both them because they are considered state of the art models and our task is a regression task to predict the relevance value in the range of 1 to 3. The extraction was done as in the previous part from the LSTM layer of the network

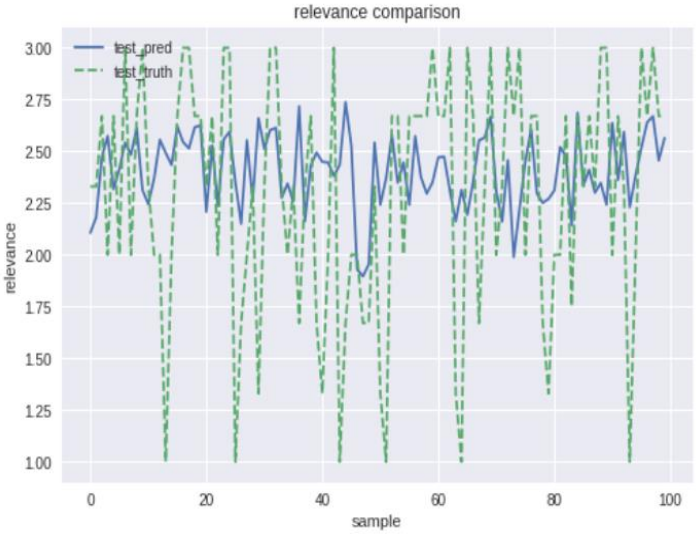
Random forest

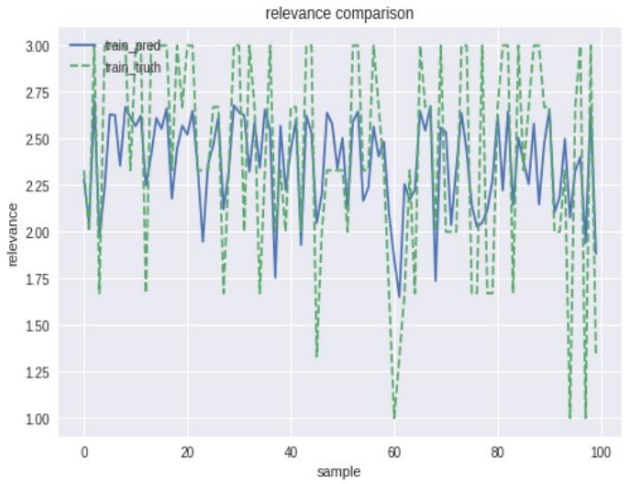
We used the sklearn random forest implementation with the following parameters: n_estimators = 15, max_depth = 10 same as before

We trained the model only on the features extracted from the network, so we had 400 features on 72K rows for the model to learn. The results are as follows: a slight overfit on the train but on the validation, we got our best result yet. So, the word embedding works better as feature extractor.

```
Running time: 230.9106481075287s
---------------------------------
Train RMSE loss: 0.4190570903812111
Train MAE loss: 0.34332484572853744
---------------------------------
Validation RMSE loss: 0.47434488994457163
Validation MAE loss: 0.38410247922725393
---------------------------------
Test RMSE loss: 0.5257103274194636
Test MAE loss: 0.427543274099431
---------------------------------
```







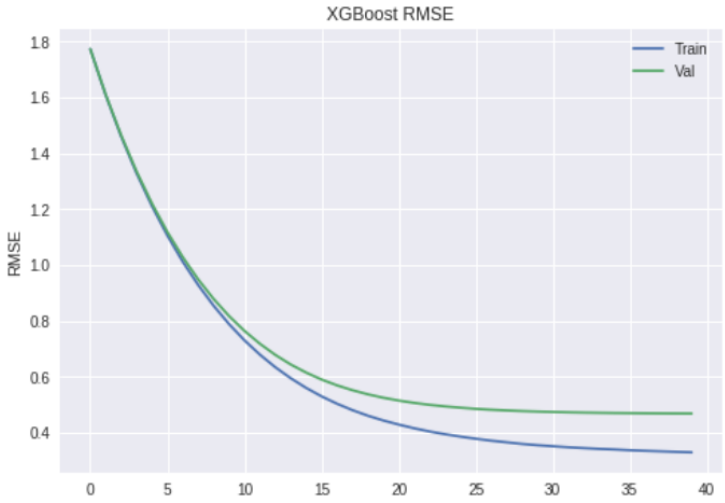
XGBRegressor

We used the XGB package implementation of XGBRegressor with the following parameters:

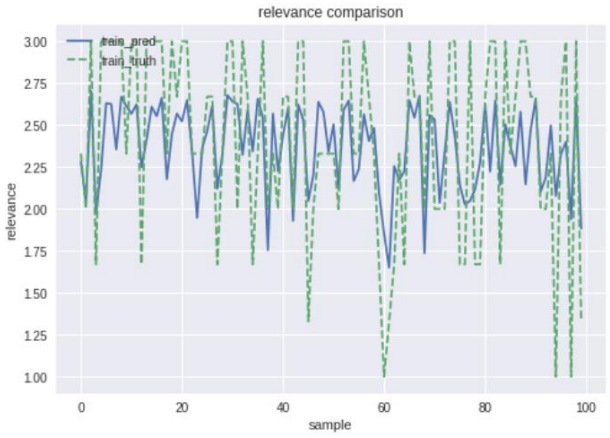
We trained the model only on the features extracted from the network, so we had 400 features on 72K rows for the model to learn. The results are as follows:

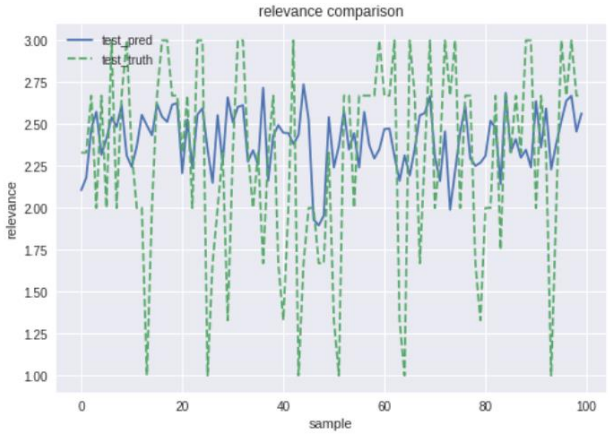
"Hooray" the XGBRegressor with feature extraction from the word embedding has the best score on the validation set from all other models! Even when again after the tuning we have a slight overfit we couldn't solve if we wanted the model to learn.

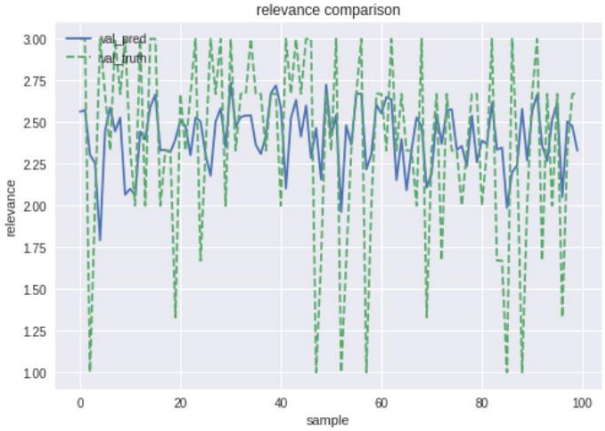
```
Running time: 218.33461737632751s
---------------------------------
Train RMSE loss: 0.32928501792424447
Train MAE loss: 0.2715796981882212
---------------------------------
Validation RMSE loss: 0.46826442505436
Validation MAE loss: 0.3803850449089163
---------------------------------
Test RMSE loss: 0.52798676365223
Test MAE loss: 0.4320116999517225
```



The predictions against the real values can be seen here for each dataset:







Result summary table:

Model type	Fit Runtime in sec	Train RMSE	Val-RMSE	Test-RMSE	Train MAE	Val-MAE	Test-MAE
Naïve model	0.00787	0.526	0.523	0.527	0.432	0.430	0.432
Character LSTM	1089.559	0.484	0.513	0.537	0.393	0.414	0.435
xgbRegressor_char	205	0.390	0.488	0.529	0.324	0.397	0.435
randomForestRegressor_char	270	0.470	0.498	0.531	0.389	0.410	0.436
Word LSTM	1253	0.418	0.418	0.529	0.338	0.340	0.430
xgbRegressor_word	218.334	0.329	0.468	0.527	0.271	0.380	0.432
randomForestRegressor_word	230	0.419	0.474	0.525	0.343	0.384	0.427

Final remarks

In this assignment we got a chance to use ANN with LSTM models word embedding. We saw and learned how to use pre-trained word embedding using GloVe and created our own character level embedding after a lot of effort. We were very surprised to see that even the relatively simple implementation we did for the char embedding gave pretty good results (of the embedding) even without a lot of epochs to the network.

Sadly, our deep learning approaches using the new Siamese LSTM architecture didn't achieve significant results on the test set compared to a naïve base line we did although the results were a little better on the validation set, this might be due to the embedding process or the relatively simple implementation of the network or the validation being a little off.

In the part of the feature extraction for the first time we achieved better results with ML models (on the test set, and in the character level on the validation as well) than just ANN model (or maybe we just used the wrong models till now?), and in this work, we saw the importance of hyperparameter tuning when at first with some hyper parameters values the classic machine learning models couldn't learn at all.

We also learned the importance of writing 'checkpoints' which validate the flow of the work, and thus reduce bugs that 'were not' supposed to happened, and also a bit of tensorflow code which help to the understanding of the mechanism behind keras has a graph tensors, which could help us in the future to implement more complex ANN architecture which is harder by just the high level keras library understanding.

Finally, we got a better understanding of embedding in this work which was good after the last one.