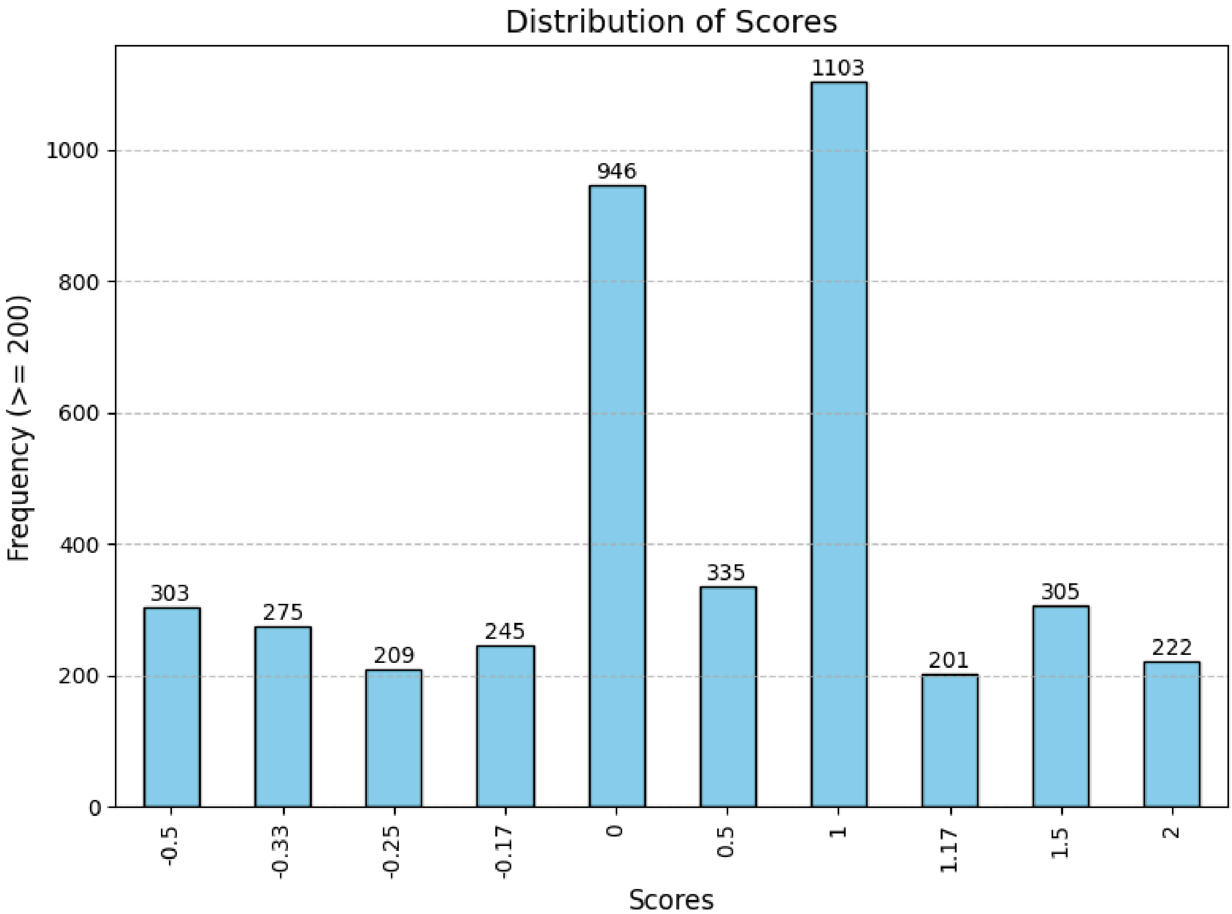
Visual Sentence Complexity Prediction

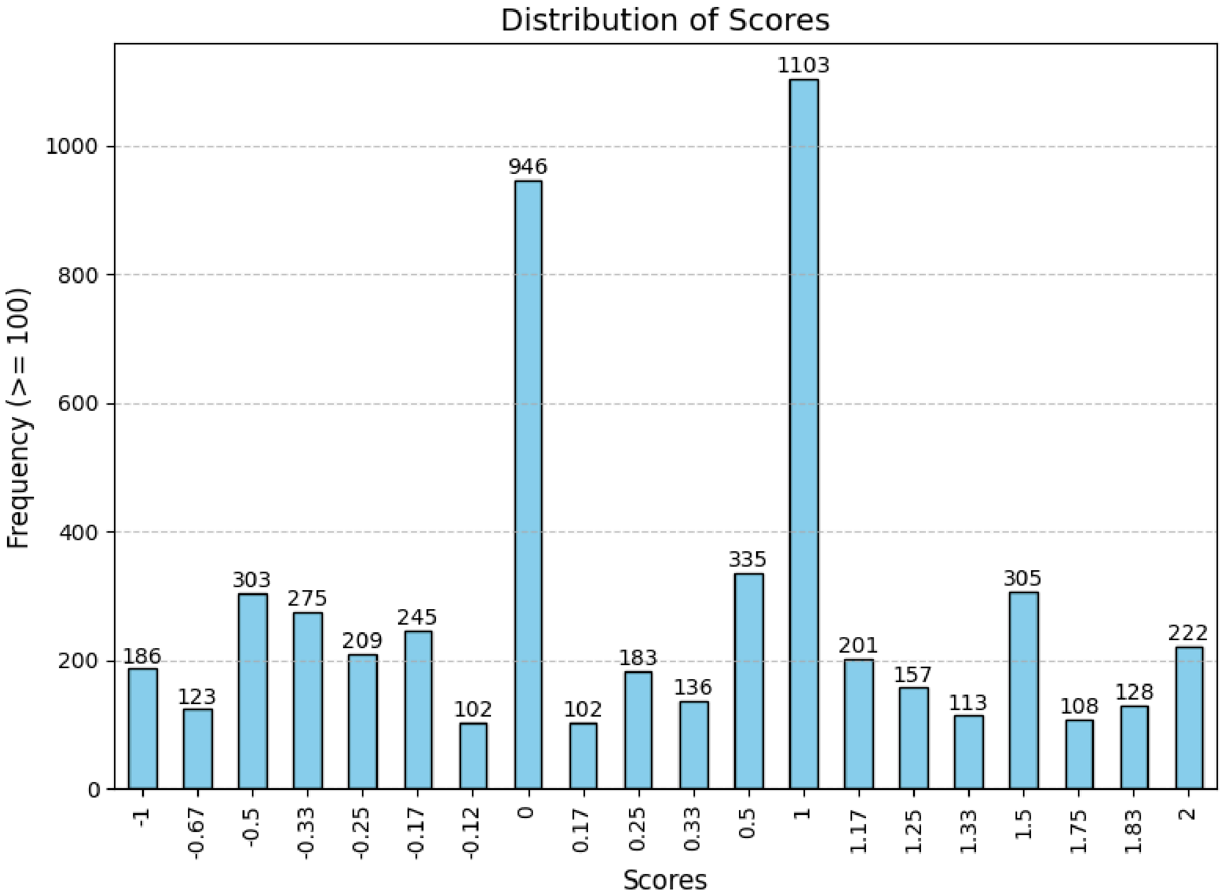
Adela-Nicoleta Corbeanu, NLP
December 2024

Visual complexity refers to an image and it measures how elaborate it is, how much it stimulates the observer, etc. The current task provides short sentences describing images, which need to receive a score inside **`[-1;2]`** indicating the visual complexity of each image.

Data Analysis

The dataset is split into train (8K sentences), validation (500 sentences) and test (500 sentences) data. By looking at the distribution of scores of the train data, an interesting pattern may be observed. It is enough to plot only the scores that are given to at least 100 or 200 sentences:





One observation that can be made is that the scores `0` and `1` are greatly common compared to the others.

Besides score distribution, another aspect that could be of interest are “out of vocabulary” words, meaning words that appear in test data and do NOT appear in train data. The results below are obtained after lowercasing everything and removing all punctuation:

Number of words in test data not in train data: *437*
Sample missing words: *['flesh', 'lwith', 'chin', 'intelligence', 'peek', 'gel', 'king', 'employees', 'problems', 're']*

If we apply stemming, we are left with:

Number of words in test data not in train data: *357*
Sample missing words: *['explain', 'flesh', 'lwith', 'chin', 'widexli', 'gel', 'rippl', 'king', 'georg', 're']*

Preprocessing

I have experimented with three preprocessing techniques:

- **Lowercasing:** converting all data to lowercase.
- **Punctuation removal:** keeping only alpha-numeric characters.
- **Stemming:** transforming each word into its "stem". For this, I used the Porter Stemming algorithm that is easily accessible in Python via the `nltk.stem` submodule.

The table below contains performance comparisons measured by Spearman correlation on validation data. The experiments have been conducted using the same model and vectorizer for all cases, particularly a `KNN` (with `K = 81`) along with a basic `TF-IDF` vectorizer.

Lowercase	Punctuation removal	Stemming	Spearman correlation
✔	✔	✔	0.6205
✘	✔	✔	0.6204
✔	✘	✔	0.6198
✔	✔	✘	0.6201
✘	✘	✔	0.6204
✘	✔	✘	0.6200
✔	✘	✘	0.6196
✘	✘	✘	0.6180

Vectorization

After preprocessing the data, the next step consists of converting all textual terms into numerical vectors. For this, I have used a frequency-based vectorizer, particularly **`TF-IDF`** vectorizer, easily accessible in Python. Its best results have emerged with the following configuration:

- Limit the vocabulary size to the top `7000` terms
- Exclude terms that appear in more than `85%` of the entries
- Exclude terms that appear in less than `2` entries
- Consider unigrams and bigrams
- Apply `L2` normalization
- Remove English stopwords

Additionally, I have also tried `Count Vectorizer` but it has performed worse than `TF-IDF` in all cases.

Data Augmentation

I have tested and compared multiple data augmentation methods using a **`KNN`** model with **`K = sqrt(N)`**, where **`N`** is the quantity of data (default+augmentations). They are as follows, ordered from best to worst results:

1. Duplicate the sentences where the score is neither **`1`** nor **`0`**
2. Duplicate the sentences where the score is in **`[-1 ; -0.3]`** ∪ **`[1.3 ; 2]`**
3. Take two random sentences **`S1`** and **`S2`**. Take the first half of **`S1`** and the second half of **`S2`** and concatenate them. To obtain the score, compute **`(score[S1] + score[S2]) / 2`**. Do this `5000` times.
4. Take two random sentences **`S1`** and **`S2`**. Take the first half of **`S1`** and the second half of **`S2`** and concatenate them. To obtain the score, compute **`(score[S1]*len(S1) + score[S2]*len(S2)) / (len(S1) + len(S2))`**. Do this `5000` times.
5. Duplicate all sentences

Additionally, I have also trained the models on validation data at the end.

K-Nearest-Neighbours (KNN)

The best performance I recorded was achieved by a **`KNN`** model. Since the algorithm is based on finding the closest **`K`** neighbours for each entry, there are three main decisions to be made:

1. The number **`K`** of neighbours to consider
2. How to calculate the distance between two entries
3. How an entry is influenced by its neighbours

For each of them, I have experimented with multiple values/methods.

First, the number **`K`** of neighbors. I have noticed that the optimal Spearman correlation tends to be obtained for **`K ≅ sqrt(N)`**, where **`N`** is the number of entries in train data. I have tried all values in the interval **`[N - 100 ; N + 100]`** and some of them are showcased in the table below, with **`81`** being the best (from a Spearman perspective, which is our main interest in this task).

	Spearman	**MSE**	**MAE**	**Kendall**
K = 50	0.615	0.423	0.526	0.437
K = 81	0.623	0.426	0.533	0.447
K = 100	0.618	0.433	0.539	0.442

The table below shows correlations for three types of metrics, the best one identified being cosine. The experiments have all been conducted with **`K = 81`**.

Metric	**Spearman**	**MSE**	**MAE**	**Kendall**
cosine	0.623	0.426	0.533	0.447
euclidean	0.617	0.430	0.533	0.442
Minkowski	0.616	0.430	0.532	0.440

The last aspect for our **`KNN`** is the weights system, particularly whether it is better to give all neighbors equal weights (uniform weights) or to give closer neighbors a bigger influence (distance-based weights).

The table below (registered for `K = 81` and `metric = 'cosine'`) shows that distance-based weights perform slightly better in all cases:

Weights	Spearman	MSE	MAE	Kendall
distance	0.623	0.426	0.533	0.447
uniform	0.605	0.437	0.539	0.429

Support Vector Regression (SVR)

My second approach consists of a **SVM** regressor, for which I experimented with various values for mainly three parameters:

- kernel type
- regularization parameter **C**
- kernel parameter gamma for **RBF** kernel

A vital role in **SVR** is played by the kernel function. In this task, **RBF** has proven to be the most effective and flexible one:

Kernel	Spearman	MSE	MAE	Kendall
rbf	0.618	0.416	0.510	0.439
linear	0.592	0.450	0.523	0.417
poly	0.597	0.464	0.568	0.425
sigmoid	0.557	0.496	0.548	0.395

In **SVR**, the **C** parameter controls the balance between minimizing training error and maintaining model simplicity. Some common values for **C** can be found in the table:

C value	Spearman	MSE	MAE	Kendall
0.1	0.584	0.457	0.555	0.409
1	0.618	0.416	0.510	0.439
10	0.592	0.431	0.521	0.418
100	0.593	0.431	0.521	0.419

Lastly, since **RBF** gave the best performance on validation data, I took it further and experimented with multiple values for its gamma parameter, which controls how “far” the influence of each data point goes.

Gamma	Spearman	MSE	MAE	Kendall
0.01	0.562	0.482	0.578	0.391
0.1	0.605	0.432	0.524	0.427
1	0.618	0.416	0.510	0.439
scale	0.618	0.416	0.510	0.439
auto	0.532	0.658	0.702	0.368