Sentence Pair Classification

Task Summary.

The competition’s task consists of classifying pairs of Romanian Language sentences into 4 classes. The meaning of the 4 classes is, however, unknown.

Observations.

- The dataset is heavily imbalanced towards classes 2 and 3. The number of samples from classes 0 and 1 totaling under 10% of the full dataset.
- Sentences are not exclusively written using the 31 * 2 characters from the Romanian Alphabet, also using characters from the Chinese Alphabet, Greek Alphabet, Mathematic Symbols etc.
- All the sentences are affirmative (they all end in the dot ‘.’ character)
- The writing style is not homogenous (there are at least 3 variations of the character ‘ș’, characters such as ‘½’ used in pair with mathematical representation “1/2”.
- Despite that, the texts seem to follow a standard norm in terms of letter capitalization (the first letter is only capitalized in names, etc.)

Preprocessing techniques.

The first step in solving the task was finding a way to preprocess the text so the subsequent models can learn more easily. Some of the techniques tried, include:

- Lowercasing all the characters
- Removing all numbers, and replacing them with a special token
- Romanizing all characters (replacing each character with their closest pair in the latin alphabet)
- Converting numbers to their Romanian language representation
- Removing all punctuation
- Stemming

- Transforming mathematical fractions and exponent symbols to their digit form.

Lowercasing, number removing and punctuation removing seemed to negatively affect the result, while converting numbers to their Romanian language representation and stemming did not seem to make much of a difference.

For stemming, I used Snowball stemmer from the NLTK library. Stemming is a rule-based process for eliminating word affixes.

Vectorizing data.

After preprocessing the natural language texts, they must be transformed into a numerical representation to be later fed to our model of choice.

For this step, I used TF-IDF Vectorization technique provided by Scikit-Learn. Prior to using the vectorizer, I concatenated the sentences in each training sample, either using a delimiting token or not (this step did not make any notable difference).

The model analyzes features chosen by the user, and attributes a score to each of them, signifying the correlation of that feature with the class label of that example.

The score for a sample **s** and a feature **f** is computed using the following formula:

TFIDF_SCORE(f,s,d) = freq(f,s)/len(s)*(log(len(d)/no(f,s))+1)

Where **d** is the full training set.

The Scikit-Learn function provides a few parameters that can be tuned to better suit the task, most notable being:

- Lowercase - (whether or not to lowercase the text prior to computing scores)
- Analyzer (‘word’, ‘char’ or ‘char_wb’) - indicates which way should the vectorizer choose its features (words, chars, or char sequences that are completely included in words)
- Ngram_range - sets an interval of lengths the chosen features must have, it can be used to capture smaller or higher positional dependencies.

- Norm – The output vector from applying the vectorizer is normalized using a norm (can be ‘l1’ or ‘l2’, for example) to provide better numerical stability for the model.
- Max_features – can be chosen to limit the number of chosen features, using only the ones with the highest score.

The TFIDF vectorizer provides a sparse representation of data, in the shape of a sparse matrix where each line represents a document, and each column a feature in that document.

Although some models are able to learn from such representation, for some of them (notably Deep Learning models) it might be needed to convert the representations to a dense matrix form.

LinearSVC.

The Support Vector Machine (SVM) models are a family of models which work by trying to find a hyperplane that separates the vectors from different classes, after projecting them to a higher dimensional space.

The LinearSVC model is a faster implementation of SVC with linear kernel.

The kernel function in the SVC is used to transform data into the desired form for separation. The linear kernel is used when the data is presumed to be linearly separable.

It uses a One-vs-Rest approach, in our case training one classifier for each class.

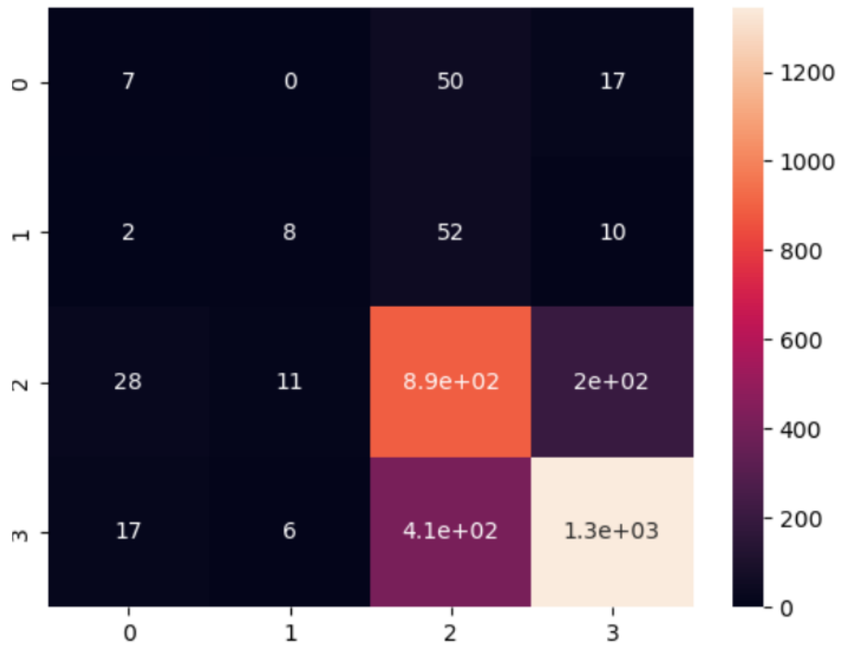
The LinearSVC provided by sklearn also has some hyperparameters which can be tuned to achieve better results. The ones I used were:

- The parameter C. It controls the model’s behavior regarding choosing a wider margin for the separation hyperplane over having less misclassified samples. For this task, a higher value of the C parameter achieved better results in practice.

- Penalty norm. It is used to choose the form of regularization used by the model. It can be ‘l1’ for linear regularization or ‘l2’ for squared one. When choosing ‘l2’ the squared sum of weights is added to the loss.

Analyzer	Ngram_range	C	F1-Score	F1-Score class 0	F1-Score class 1	F1-Score class 2	F1-Score class 3
Char	1 - 5	50	0.444	0.109	0.164	0.702	0.802
Char	1 - 3	10	0.454	0.121	0.244	0.668	0.784
Word	1 - 1	10	0.413	0.044	0.211	0.650	0.745
Word	1 - 3	5	0.450	0.134	0.204	0.696	0.765

The confusion matrix for the best scoring model (the first one) is:



Class	Precision	Recall
0	0.13	0.09
1	0.32	0.11
2	0.64	0.79
3	0.85	0.76
Overall	0.48	0.44

The overall accuracy of the model is 0.73.

Fully Connected Neural Networks.

This approach consists of building a Neural Network architecture that is able to classify the input vectors into the 4 classes.

As stated before, this approach requires the input vectors to be dense, so we cannot afford to use all the features provided by the TFIDF Vectorizer. For this reason, only a few thousands were used.

The best architecture used consisted of 3 layers, 2 hidden ones and a final one with an output dimension of 4 (the number of classes). After the 2 hidden layers, I used the **ReLU** activation function and, after the first one, a **dropout layer** with a rate of 20%.

Each fully connected layer is essentially a matrix of weights. In each model passing, for each layer, its output is computed using the formula:

$$O = W * I + B$$

Where:

- W is the weight matrix
- I is the input vector
- B is the vector of biases
- O is the output vector

The activation functions, (such as ReLU) are used to help the model compute nonlinear functions, which would not be possible with matrix multiplications alone.

The formula for the ReLU activation function is: $ReLU(x) = max(0,x)$.

One common problem in deep neural networks is overfitting, the model learning more about the data than it should, not being able to extrapolate on new samples. To prevent overfitting, we can use many approaches (simplify our model architecture, adding data samples, etc.) and one of them is the introduction of Dropout.

The dropout step, means that a percentage of the neurons of a layer do not participate to the computation of the result at one step. (The neurons are randomly selected at each step).

As the loss function, I have opted for Categorical Cross Entropy Loss, which is an accurate way of computing a loss for a multi-class classification problem.

The formula for this loss function is the following:

$$-\sum_{i} t_i \, log \, p_i$$

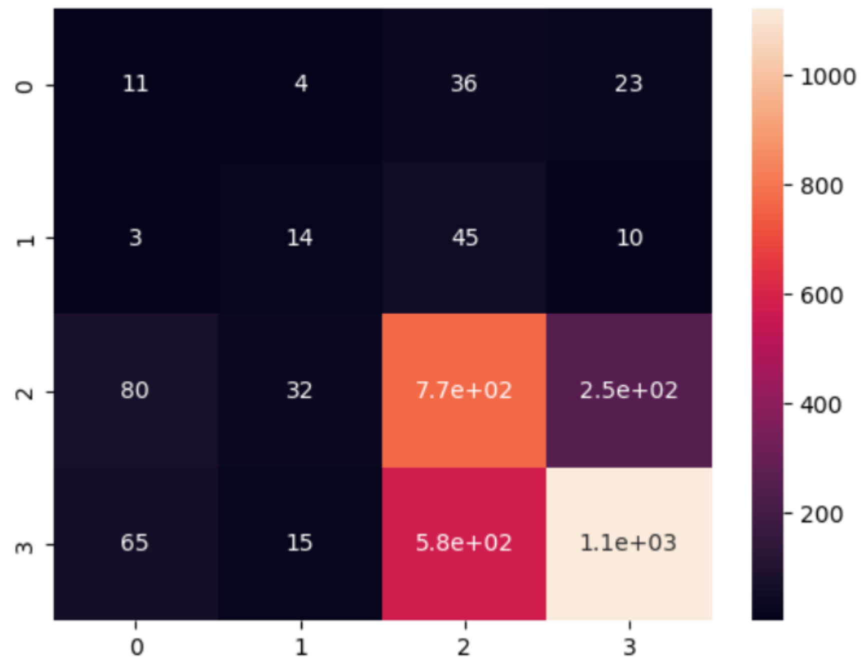
Where i is a class in our problem, ti is the true probability of that class, and pi is the predicted probability of that class.
As an optimizer, I have opted for the Adam optimizer without callback functions. It helps the model converge quicker.
The final architecture of the model is the following:

```
(fullyConnected): Sequential(
    (0): Linear(in_features=24000, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=128, out_features=32, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=32, out_features=4, bias=True)
  )
```

For the neural networks, I used the PyTorch framework.

Word features	Char features	Learning Rate	Epochs	Layers	F1-score	F1-0	F1-1	F1-2	F1-3
12000	12000	0.001	10	3	0.40	0.06	0.23	0.62	0.71
10000	14000	0.0001	10	3	0.39	0.10	0.26	0.50	0.72
12000	12000	0.0001	5	3	0.38	0.10	0.25	0.47	0.69

The confusion matrix for the best model (the first one) is:



Class	Precision	Recall
0	0.07	0.15
1	0.22	0.19
2	0.54	0.68
3	0.80	0.63
Overall	0.40	0.41

The accuracy of the model is 0.63.

Counteracting the class imbalance.

Every SKlearn model or torch.loss has a possibility to provide percentages for each class in the training set. These percentages are used when computing the loss function.

They are chosen inversely proportional to the frequency of a class, in that way, the classes seem to be “penalized equally”.

Final Remarks.

The final 2 models chosen were one neural network and one SVC model. They performed very similarly on the test set, at an f1 score of about 0.651.