OR 610 Final Project: U.S. Patent Phrase to Phrase Matching

Team Members: Julie Carey, Neb Hrnjez, Derek Margulies, Chris Smith, Ben Zevin

Introduction

For the OR610 term project this group has decided to take on a Kaggle competition hosted by the U.S. Patent and Trademark Office centered around solving a Natural Language Processing problem.

The requirement of this competition is to design, implement and train a model to output a similarity score between two phrases between documents of the same context. Similarity scores are broken down in 5 main categories:

- 1 = Very Close Match, almost and exact match with some slight differences
- 0.75 = Close Synonym, e.g., “mobile phone” and “cellphone”, this would also include abbreviations
- 0.5 = Synonyms that don’t have the same meaning
- 0.25 = Somewhat related, e.ge two phrases are in the same high-level domain but not synonyms. E.g., “dog” and “cat” are both “animals”
- 0.0 = Unrelated

Ultimately evaluation of the model will be done by comparing the predicted and actual similarity scores using the Pearson correlation coefficient.

Motivation

The motivation behind this problem stems from the nature of the U.S. Patent and Trademark Office’s work. Every time a patent is submitted it undergoes a vigorous examination process to determine whether an application meets the requirements of a claimed invention, the scope of that claim and if that technology exists already for the claimed invention. This process is repeated twice with the examination spanning the entirety of the U.S. Patent archives (some 11 million patents).

If a model can be trained to determine the similarity between existing patents and new applications this would both reduce the number of man hours required by the patent office to do its job and also improve depth of patents examined during the search leading to better outcomes.

Data Sources & Exploration

Source

As previously mentioned, the idea and consequently the dataset are from a Kaggle competition hosted by the U.S. Patent and Trademark Office. Upon further inspection it seems that this dataset is part of or in some way related to the “Google Patent Phrase Similarity Dataset” by Google, but further research is needed to determine the extent.

The dataset and competition can be found here:

https://www.kaggle.com/competitions/us-patent-phrase-to-phrase-matching/overview (https://www.kaggle.com/competitions/us-patent-phrase-to-phrase-matching/overview)

The download consists of three files: sample_submission.csv, test.csv and train.csv. The train.csv and test.csv datasets will be used to train and test the model respectively. These two datasets share the following columns:

- id: a unique identifier for each pair of phrases.
- anchor: one of the two words or phrases that will be compared to generate a similarity score.
- target: the other word or phrase to be compared for generation of a similarity score.
- context: a category identifier for the two words from the CPC classification (version 2021.05) indicating the subject to which the patent relates

The following only appears in the train.csv dataset:

- score: a score based on how similar “anchor” and “target” are in the relative context, broken down into 5 possible scores (0, 0.25, 0.5, 0.75 and 1).

The competition requires a score to be generated (predicted) for the test.csv dataset using the model created which will then be compared by officials to actual scores.

Exploration

This data requires very little cleaning as this is part of a government sponsored Kaggle competition with a focus on solving the problem not data preparation. The important component of data exploration for this project is to understand the nature of data and how best to extract it for the problem at hand.

Size:

For the train.csv dataset there is 36473 datapoints.

For the test.csv there is 36 datapoints.

id:

This column does not serve any predictive purpose in terms of generating a similarity score but only exists to identify each instance of data for later use. A requirement for the competition is that submission of output must be in the following form:

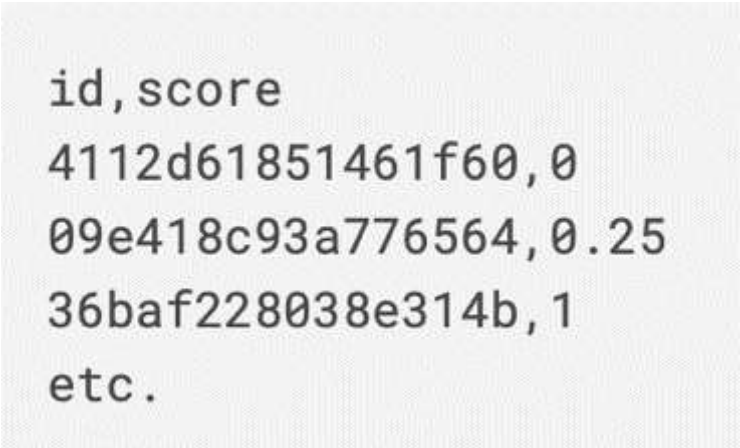


Figure 1: Example of required format for competition submission file

The “id” column ensures that predicted scores are compared to the correct actual scores and the data can be followed throughout the model.

anchor:

This column is used to generate a similarity score by determining its semantic similarity with the phrase in the "target" column. Looking specifically at the train.csv dataset we notice that while there are 36,473 data points there are only 733 unique values.

```
count                                36473
unique                                 733
top       component composite coating
freq                                   152
Name: anchor, dtype: object
```

Figure 2: traincsv.anchor.describe() output

Further examination indicates that these words seem to be more general than the words in the "target" column, such as the example seen in Figure 3:

	id	anchor	target	context	score
0	37d61fd2272659b1	abatement	abatement of pollution	A47	0.50
1	7b9652b17b68b7a4	abatement	act of abating	A47	0.75
2	36d72442aefd8232	abatement	active catalyst	A47	0.25
3	5296b0c19e1ce60e	abatement	eliminating process	A47	0.50
4	54c1e3b9184cb5b6	abatement	forest region	A47	0.00

Figure 3: Generalized words in "anchor" vs. more specific words in "target"

This column contains both single words and short phrases of words to express a concept or idea.

target:

This column is very similar to the "anchor" column in the sense that it is the same structure and that it is one part of the comparison needed in generating a similarity score. Looking specifically at the train.csv dataset there are 29,340 unique words out of a possible 36,473 datapoints.

```
count           36473
unique          29340
top       composition
freq               24
Name: target, dtype: object
```

Figure 4: traincsv.target.describe() output

Figure 4: traincsv.target.describe() output

As mentioned previously while the phrases in the "anchor" category are more general, those in the "target" column seem far more varied as seen in figure 3. Again, the "target" column contains both single words and short phrases of words.

context:

This column contains a context identifier for the two phrases in the same row. The identifier come from a list called the cooperative patent classification system or CPC. This list has been jointly developed the European Patent Office and the U.S. Patent and Trademark Office. In this dataset there are only two components to context: section (one letter A to H and Y) and class (two digits). Each section relates a general category for the patent i.e. A: Human Necessitates, D: Textiles and H: Electricity. The class represents a more exact category i.e. A01: Agriculture; forestry; animal husbandry; trapping; fishing.

When looking at the train.csv dataset for the 36473 datapoints there are only 106 unique context identifiers as seen in Figure 5.

```
count     36473
unique      106
top         H01
freq       2186
Name: context, dtype: object
```

Figure 5: traincsv.context.describe() output

This however changes the number of unique “anchor” and “target” phrases as a single phrase can apply to multiple context identifiers. This increases the number of unique phrases when looking at “anchor” and “target” together with “context” respectively. As seen in Figure 6 the number of unique “anchor”/”context” phrases and “target”/”context” phrases increase to 1,699 and 35,850 respectively.

```
anchor          context        target                    context
abatement       A47        21  1 amino 2 methoxyethane   C07        1
                A61         3  1 azabicyclo              G01        1
                A62         1  1 bromopropane            C09        1
                C01         1                            C10        1
                F16         1  1 methoxy 2 aminoethane   C07        1
                           ..                                      ..
wiring trough   F16        27  zone bells                G08        1
                H02        18  zoom factor               G06        1
wood article    B05        28                            H04        1
                B27         1  zoom in capacity          G03        1
                B44        27  zoster virus infection    C12        1
Length: 1699, dtype: int64     Length: 35850, dtype: int64
```

Figure 6: Unique combinations when combining “anchor” and “target” with “context”

score:

This column only appears in the train.csv dataset as this is the response variable that the model is trained against. Examining the column within the train.csv dataset, the following distributions are noticed.

```
0.50    12300
0.25    11519
0.00     7471
0.75     4029
1.00     1154
Name: score, dtype: int64
```

Figure 7: Distribution of datapoints by score values

These values were previously explained, 1: indicates a very close match and 0 indicates no match at all. The majority of datapoints occur at the lower end of similarity score between 0 and 0.5 where the two phrases do not match very well. The smallest group is the 1.0 score group or very close match, this is to be expected as the criterion for this score is almost an exact match between the words of which there are few.

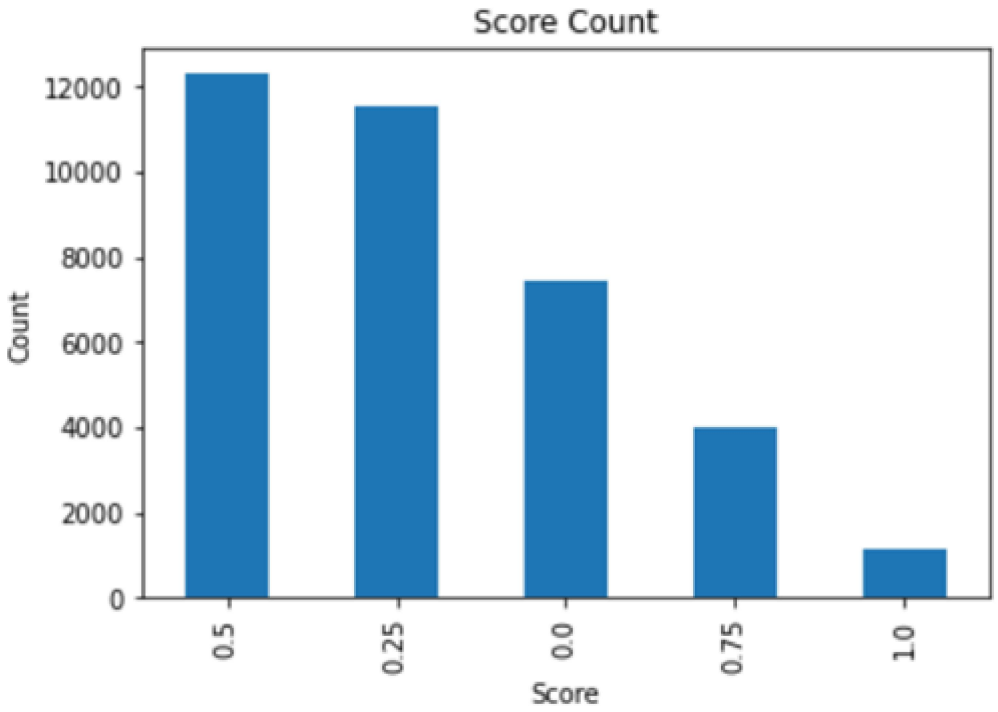


Figure 8: Graph showing score counts based on score value

Further Exploratory Analysis: PCA

Because of the high dimensionality of the model, we were interested in taking an unsupervised approach to clustering for the problem in order to gain a better high-level understanding of similarities between semantics. To do this we took each of our sentences and using PCA projected our sentences into both the 2 and 3 dimensional spaces. After this we viewed our embeddings with the hope that similar entities would tend to cluster in similar spaces and we could throw some time of clustering algorithm (Kmeans, DBScan) at the problem and make a prediction that way

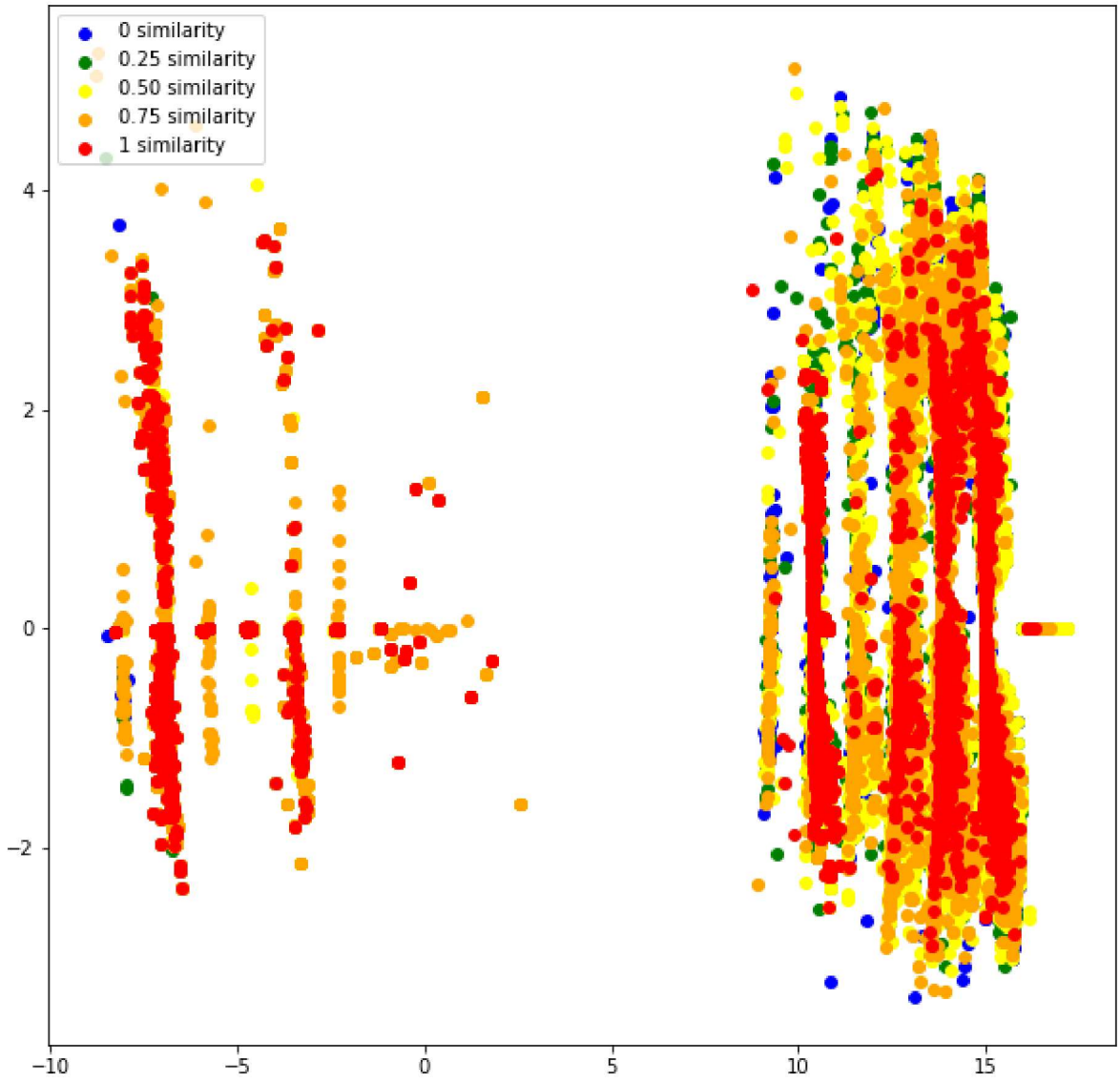


Figure 9: PCA Results with Scores Shown

After performing our PCA and looking at the data we can see that regardless of the target/anchor sentences similarity they tend to be clustered in the same space meaning that this data is probably suited towards a solution in a higher dimensional space/more complex solution than simple unsupervised clustering

Project Logic

From OR610 lectures, a key component in NLP problems is how you vectorize the words, phrases or sentences fed into the model. There are two ways to deal with language inputs since the model needs to predict the semantic similarity between words based on context and one-hot encodings which are only vectorized as an orthogonal representation.

Prior to generating the embedding matrix, the problem must be examined from a high-level logic perspective. In the simplest terms, we need to compare the “anchor” and “target” to generate a similarity “score” while also considering the “context” of the words we are comparing. This “context” component adds a level of complexity to our problem, without it pre-trained vectors such as GloVe or fastText WIKI could be used as the similarity would be based only on the words in respect to their use in English language. This approach will be discussed in detail in later sections. The “context” column changes the meaning of “anchor” and “target” inputs. Looking at Figure 6 the “anchor” word “abatement” has “context” classes A (Human Necessities), C (Chemistry and Metallurgy) and F (Mechanical Engineering). Any model designed needs to take this feature into account to be truly successful.

Another important thing to consider when designing our model and more specifically our embedding matrix is future use cases of this model. This problem can be summarized as “overfitting” where the model essentially only works on inputs it has seen before. This will essentially be a function of how inputs are approached in terms of how “context” will be applied to the phrases.

Logical Approaches:

Treating each unique row as an individual embedding:

One possible approach is to treat each unique row as an embedding, in such that every unique combination of “anchor” and “context” and “target” and “context” gets its own embedding, this is illustrated in the figure below:

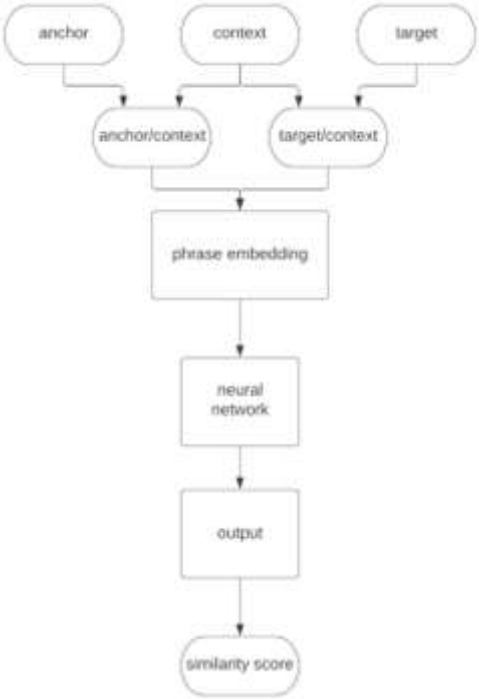


Figure 10: Unique phrase/context model logic

In this model, the effect of the “context” column is captured for each unique combination with the phrases in “anchor” and “target. Looking again at figure 6, “abatement” in context “A47” and “abatement” in context “A61” would have unique embedding tensors. This also has the effect of simplifying inputs as we are combining the two or more-word phrases in “anchor” and “target” into a single word (from figure 3: “abatement of pollution” would be entered together instead of split as “abatement”, “of” and “pollution”)

Implementation of this can be seen in the attached notebook “OR610_Draft_Embedding_Code1” (Code was done as user input but can easily be automated). The code is as follows:

1. For the train.csv dataset, combine “anchor” and “context” and “target” and “context” and then drop the component columns, as seen in figures 10 and 11

	id	anchor	target	context	score
0	37d61fd2272659b1	abatement	abatement of pollution	A47	0.50
1	7b9652b17b68b7a4	abatement	act of abating	A47	0.75
2	36d72442aefd8232	abatement	active catalyst	A47	0.25
3	5296b0c19e1ce60e	abatement	eliminating process	A47	0.50
4	54c1e3b9184cb5b6	abatement	forest region	A47	0.00
...	...	...	...	...	...
36468	8e1386cbefd7f245	wood article	wooden article	B44	1.00
36469	42d9e032d1cd3242	wood article	wooden box	B44	0.50
36470	208654ccb9e14fa3	wood article	wooden handle	B44	0.50
36471	756ec035e694722b	wood article	wooden material	B44	0.75
36472	8d135da0b55b8c88	wood article	wooden substrate	B44	0.50

Figure 11: Original dataset

	id	score	anchorcontext	targetcontext
0	37d61fd2272659b1	0.50	abatementA47	abatement of pollutionA47
1	7b9652b17b68b7a4	0.75	abatementA47	act of abatingA47
2	36d72442aefd8232	0.25	abatementA47	active catalystA47
3	5296b0c19e1ce60e	0.50	abatementA47	eliminating processA47
4	54c1e3b9184cb5b6	0.00	abatementA47	forest regionA47
...	...	...	...	...
36468	8e1386cbefd7f245	1.00	wood articleB44	wooden articleB44
36469	42d9e032d1cd3242	0.50	wood articleB44	wooden boxB44
36470	208654ccb9e14fa3	0.50	wood articleB44	wooden handleB44
36471	756ec035e694722b	0.75	wood articleB44	wooden materialB44
36472	8d135da0b55b8c88	0.50	wood articleB44	wooden substrateB44

Figure 12: Combined column dataset

2. Combine “anchorcontext” and “targetcontext” into a single long vocabulary and create an index.
3. Determine vocabulary size and an embedding dimension.
4. Create an embedding layering using nn.Embedding
5. Create an embedding matrix by running your encoded words through the embedding layer as seen below:

```
tr1_emb = uniquewrd_emb()

Which dataset are you creating an embedding of unique words for?
tr1
                     id  score    anchorcontext             targetcontext
0      37d61fd2272659b1   0.50     abatementA47  abatement of pollutionA47
1      7b9652b17b68b7a4   0.75     abatementA47          act of abatingA47
2      36d72442aefd8232   0.25     abatementA47         active catalystA47
3      5296b0c19e1ce60e   0.50     abatementA47     eliminating processA47
4      54c1e3b9184cb5b6   0.00     abatementA47           forest regionA47
...                 ...    ...              ...                       ...
36468  8e1386cbefd7f245   1.00  wood articleB44          wooden articleB44
36469  42d9e032d1cd3242   0.50  wood articleB44             wooden boxB44
36470  208654ccb9e14fa3   0.50  wood articleB44           wooden handleB44
36471  756ec035e694722b   0.75  wood articleB44         wooden materialB44
36472  8d135da0b55b8c88   0.50  wood articleB44        wooden substrateB44

[36473 rows x 4 columns]
Enter the columns to create vocab: (seperate columns by space)  anchorcontext targetcontext
```

```
tr1_emb.shape[0]
tr1_emb

37266

tensor([[-0.0084,  0.5383,  1.2925,  ...,  1.1059,  0.8081,  0.9945],
        [ 1.6834,  1.0938,  0.3440,  ..., -0.8444,  0.5971, -0.3429],
        [ 0.3215, -0.1895, -0.3901,  ...,  0.1600,  0.0212, -1.7676],
        ...,
        [-0.9040,  2.8191,  1.3696,  ..., -0.0679,  1.5890,  3.1074],
        [-1.2836, -0.4362,  0.1808,  ..., -0.3733,  0.0210,  0.9192],
        [ 0.2080,  0.4950,  0.3874,  ...,  2.0194, -0.2519, -0.6721]],
       grad_fn=<EmbeddingBackward0>)
```

Figure 13: Embedding matrix

This model is however limited by what is available in the training set to create the embedding matrix. While a potential 37266 unique combinations might seem like enough, the competition test set will have 12000 unseen pairs of words each with a context. Consider that the English language has over 150,000 words and that the training data does not contain every CPC classification for “context”, this model would potentially be missing a large amount of data.

Additionally, similar models where “anchor” and “target” are not combined with “context” but still treated as a combined word (as explained above), are also considered but suffer from the same issue of limited vocabulary. This approach is essentially an exercise in training our own embedding layer, and for the purposes of the term project may be appropriate but when considering the competition and potential future proofing this approach would likely be insufficient.

Categorical Embeddings

Another approach is to create separate embeddings matrixes for the phrases and for the context and then concatenate them, run them through a model and generate an output. This is illustrated in the model below:

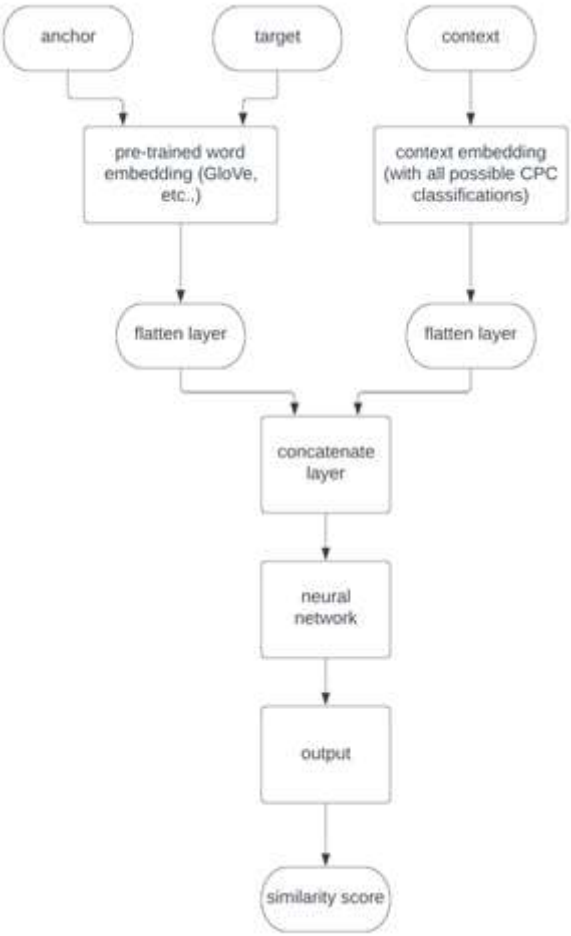


Figure 14: Categorical embedding model logic

In Figure 13, there are two key differences:

1. Each word in “anchor” and “target” will be looked at individually and run through a pre-trained word embedding layer instead of looking at the phrase a whole. This has the advantage of comparing the inputs against a much larger vocabulary leading to better use-cases in the future.

   Proof of concept of this can be seen in “OR610_Draft_Embedding_Code2” where the 50-dimensional GloVe word-embeddings are used to create a pre-trained embedding matrix and embeddings of a few words from train.csv are generated.

```
#Generate the embeddings for the following words based on index value
my_embedding_layer(torch.LongTensor([word2id['abatement']]))
my_embedding_layer(torch.LongTensor([word2id['of']]))
my_embedding_layer(torch.LongTensor([word2id['pollution']]))
```

```
tensor([[-0.2668, -0.5424, -0.1127, -0.4163, -1.8010,  0.8619,  0.7122, -0.7605,
          0.6341,  1.1180, -0.2690, -0.0735,  0.9681, -0.1955,  0.1151, -0.4148,
          0.7546,  0.4025, -0.1902, -0.6264,  0.0954, -1.1635, -1.1770, -1.0403,
         -0.7156, -0.2167, -0.0251,  0.3762,  0.9141, -0.3995,  0.0421, -0.1262,
         -0.3494, -0.7977, -0.4301,  0.4980,  1.1406, -0.8727,  1.2612, -0.0565,
         -0.4890, -0.3256,  0.4114,  0.8498,  0.0777, -0.3739, -0.1317,  0.6197,
          0.9914,  0.7029]])

tensor([[ 0.7085,  0.5709, -0.4716,  0.1805,  0.5445,  0.7260,  0.1816, -0.5239,
          0.1038, -0.1757,  0.0789, -0.3622, -0.1183, -0.8334,  0.1192, -0.1661,
          0.0616, -0.0127, -0.5662,  0.0136,  0.2285, -0.1440, -0.0675, -0.3816,
         -0.2370, -1.7037, -0.8669, -0.2670, -0.2589,  0.1767,  3.8676, -0.1613,
         -0.1327, -0.6888,  0.1844,  0.0052, -0.3387, -0.0790,  0.2419,  0.3658,
         -0.3473,  0.2848,  0.0757, -0.0622, -0.3899,  0.2290, -0.2162, -0.2256,
         -0.0939, -0.8037]])

tensor([[-5.5225e-02,  1.1566e-01,  3.0453e-01, -1.3324e-01, -1.9954e+00,
         -1.0151e-01,  9.0641e-01, -1.0230e+00,  1.5890e+00,  6.3664e-01,
          2.3525e-01, -7.8734e-04,  5.4517e-01, -2.5121e-01,  2.4846e-01,
          2.3713e-01,  7.4996e-01,  4.5571e-01, -3.9023e-01, -1.0575e+00,
          3.2863e-01, -9.2208e-01,  2.4859e-01, -1.0628e+00,  2.1975e-01,
         -1.3064e+00,  2.0489e-01,  6.6256e-01,  7.9641e-01,  6.5000e-01,
          2.2863e+00,  1.5609e-01, -9.7939e-02, -1.4865e+00, -1.3358e-01,
          1.4123e-01,  2.3711e-01, -5.4926e-01,  7.9363e-01,  1.1131e+00,
         -1.4096e+00,  7.3101e-02,  1.6536e+00,  1.0757e-01,  2.7960e-01,
          2.7203e-01, -2.8178e-01,  2.8664e-01,  1.1429e+00, -3.1253e-01]])
```

Figure 15: Generating word embeddings from pre-trained embedding layer

2. Context gets its own embedding layer which will be combined with the phrase inputs after it has passed through the layer. This “context” embedding will be trained on a dataset with all possible “context” sections and classes from the CPC classification list.

   This concatenation layer will apply the “context” information to “anchor” and “target” without losing the valuable semantic similarity information of the words. It also has the added benefit of standardizing inputs through potential models allowing **(allowing what?)**

This approach is the more complicated and difficult of the two but in likelihood will yield better initial results within the dataset while also hopefully future proofing the model to new inputs.

The next steps for this logic structure is to build upon the code in “OR610_Draft_Embedding_Code2”.

This first step is ensuring that inputs are comparable, again referencing figure 10 the first row shows “abatement” and “abatement of pollution” are to be compared. This is a relatively simple problem as it equates to treating each input like a sentence and then padding the sentence to the same length. For this there are many examples online as it is used on many other models.

Additionally, we must ensure that that during training any additional words are added to the embedding matrix while also maintaining the existing “learned” words. One example of this is complex chemical names as seen in the example below:

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/zh/y6z98n614pdd50jqljcptj340000gn/T/ipykernel_89637/2338416611.py in <module>
      3 my_embedding_layer(torch.LongTensor([word2id['of']]))
      4 my_embedding_layer(torch.LongTensor([word2id['pollution']]))
----> 5 my_embedding_layer(torch.LongTensor([word2id['hydroxyphenyl']]))

KeyError: 'hydroxyphenyl'
```

Figure 16: Missing words in pre-trained embeddings

Even with 400,000 embedded words, certain complex chemical compounds do not exist in the vocabulary. This problem also persists for certain number values. There are a number of ways to potentially deal with this; ignoring these cases due to the small percentage within the entire dataset or assigning a similarity of 0 automatically if the word is unrecognized. The second solution makes sense because if it doesn't exist within a 400,000-word vocabulary it likely isn't comparable to anything. This will be explored further during final design.

The "context" embedding layer must be created as well but again this is relatively easy as there is a limited vocabulary in comparison to the "anchor" and "target" columns.

Finally building the remaining layers, implementing a neural network and generating an output would complete this model.

Similar Approaches

During the planning of the project we sought similar models and methods to develop a base case with which to judge our approach. The hybrid NLP tutorial repo on GitHub contains a Semantic Claim model. This model creates a search engine to compare fact-checked claims using a pre-trained Bert model. The authors use the STS-B data set which contains approximately 5700 claims in the training set and 1500 claims in the test set. To develop a base case our dataset was fed through model adjusting model parameters to ensure the data fit. This first implementation follows the original architecture and does not utilize the context column in the dataset. The Kaggle training dataset was split into training and test sets using a stratified sample based on the context groupings to ensure each context appeared in the training and test sets. Training took significantly longer at 68 minutes compared to 11 minutes with the larger Kaggle dataset. The model achieved a r of 0.72 compared to the author's 0.77. The next step is to run the Kaggle test set through the tuned model. These results then serve as a base case to compare our predicted scores against.

```
Epoch 4/4
----------
train Loss: 0.9788, Pearson: r=0.7381 p=0.0000 n=27354
val Loss: 1.1635, Pearson: r=0.7191 p=0.0000 n=9119

Training complete in 68m 57s
Best loss: 1.163474 correl: 0.7191
```

Figure 17: Model training time example

Deep Learning Architectures

As for potential deep learning architectures, we plan to implement them. However, the primary type of architecture we plan to use is transformers. We plan to use the transformers package created by Hugging Face to implement these models. This package contains 108 different transformers, all of which are pre-trained and can be fine-tuned. Pre-training is precisely what it sounds like, a process done before training your model and sometimes making it unnecessary. When you initialize an empty model, the starting weights are initialized randomly, and the model's

training adjusts the importance to improve your model. Pre-training is when your model can extract patterns from a dataset to understand a basic understanding of the data before being trained on a specific dataset. For example, since our project focuses on text similarity, it would be necessary for a model to understand the English language before being exposed to our dataset. If it understands the English language before being implemented with our data, it should be able to recognize synonyms and antonyms of words which is very important to context similarity. Most models are pre-trained using an extensive dataset, such as Wikipedia.

From this Transformers package, one of the models we plan on using is a BERT (Bidirectional Encoder Representation with Transformers). A BERT extracts patterns or representations from the data or word embeddings by passing them through an encoder. The encoder is multiple transformer architecture layers stacked together. A BERT is also Bidirectional to look at text from both the left and right of the input. BERTs have two main phases or three if you want to train a BERT using your dataset. The first is pre-training which was mentioned earlier, and the second is fine-tuning. You can change model parameters such as learning rate or batch size in the fine-tuning phase. But this is also the phase where you determine the downstream task you want the model to complete, such as classification, text-generation, language translation, and even question answering.

Inside the Encoder layer of a BERT, it has two components: a self-attention layer and a feed-forward neural network. After the embeddings are passed through the encoder, they move to a decoder containing three elements: the self-attention layer, Encoder-Decoder Attention, and another feed-forward neural network. The most important difference between and the primary reason a BERT outperforms traditional RNNs, LSTMs, and GRUs is that it can understand the context of words and the meaning of words instead of just looking at the words themselves.

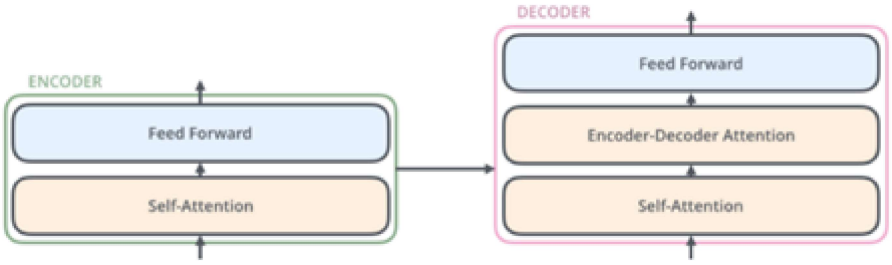


Figure 18: Transformer example

Finally, the other model we tried implementing a Manhattan LSTM (MaLSTM) model, since it's often used for text similarity. In a MaLSTM model there are two identical LSTM networks and the LSTM is passed vector representations of sentences through word2vec, in this case we used pre-trained 300 dimension word2vec vectors. The model then ouputs a hidden state encoding the semantic meaning of the sentences and the hidden states are then compared to output a similarity score. An example of the architecture can be seen below:

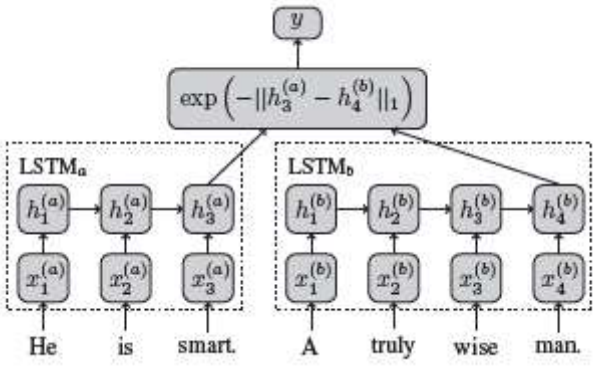


Figure 19: MaLSTM model Architecture

While this approach tends to make sense in the theoretical sense, as we are embedding sentences into a space that should make it easier for our network to pick out the differences. In practice this approach has drawbacks in that it takes a long time for the network to train. While this concern could potentially be mitigated by the use of a GPU, we believe that there are better heuristic solutions that exist and that we explore in this paper.

The BERT Model

BERT stands for Bi-Directional Encoder Representations from Transformers, developed by a team at Google led by Jacob Devlin. The original paper published in 2018 is BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. There are two versions of the BERT model, comes in two versions, a large and small. The model used in this method is the BERT Base Case. It contains 12 layers, a hidden size of 768, and 12 self-attention heads. Overall, the model contains 110 million total parameters compared to the 340 million parameters in the BERT large model. BERT was designed to predict the next sentence. It is pre-trained on a large corpus to get initial weights. When calling the model in pytorch the user is calling this pre-trained model. Fine tuning is then conducted on the user's specific dataset to obtain working weights. BERT is bi-directional, so unlike most models that work left from right, BERT works in both directions, it learns the left and right context of a word by masking elements of the sentence or input and then and predicting the masked portion. From the BERT model we output the embeddings of the input vector.

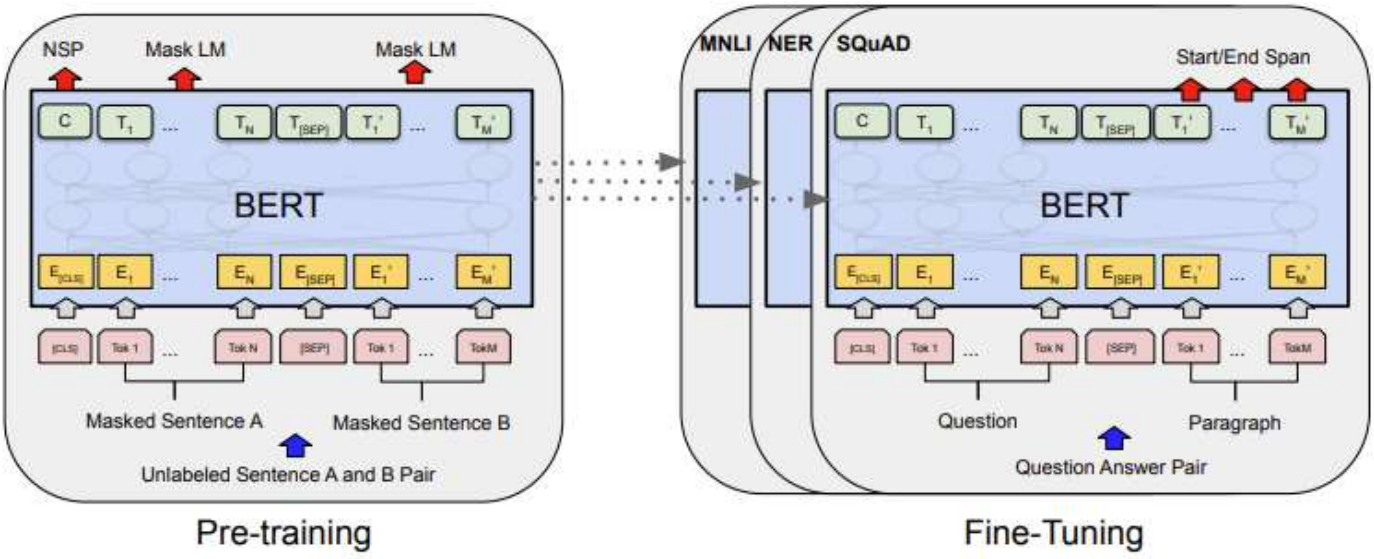


Figure 20: The BERT Model

WordPiece Tokenizer

The tokenizer implemented in the BERT model is a WordPiece tokenizer. This tokenizer is originally produced in 1980. The tokenizer not only tokenizes a single word but can break words into sub words to gain more from context. As shown in the model below the input is mapped to token embeddings. Words can be broken into sub words to learn context, i.e. airplane into "air" and "plane". Along with the embedding comes the segment embedding, this allows the model to keep track of sentences.

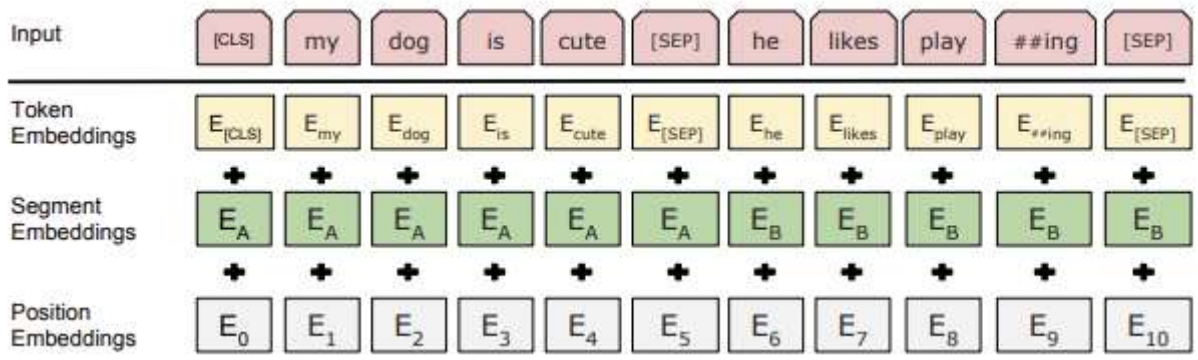


Figure 21: The BERT Model Tokenizer

From these embeddings we then use Cosine Similarity to predict the difference between the two feature vectors. Cosine Similarity works well on high-dimensional data. Cosine Similarity measures the angle between the two vectors, zero is not similar at all, one is the same. As this is a continuous distribution, we round the output to create a score in the same range as the target score for the Kaggle competition. The original implementation of the tutorial created a search engine to find articles in a dictionary that were similar to the search term. To predict a score on the anchor and target context two dictionaries were created for the anchor and target. We use the tuned BERT model to output the embeddings for the anchor and the target and then iterate through the anchor target pairs to produce a score for each pair.

MaLSTM Model Results

After running the Manhattan LSTM we ended up with an accuracy of 62% after 500 epochs and 68% after 1200 epochs. While this does imply that the model could potentially be useful if trained longer, other solutions could be implemented that are more performant/take less time to train. Interestingly we would have trained this model for longer but ran into a time limit for the Kaggle competition to train it.

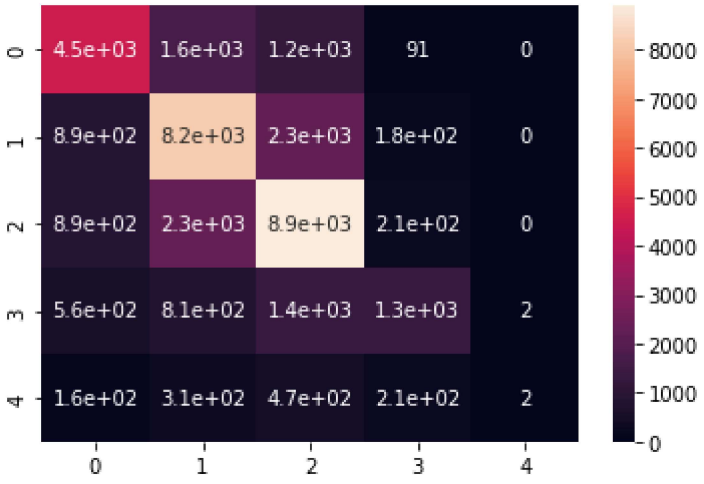


Figure 22: Confusion Matrix of MaLSTM Results

Model with Hugging Face Functions

Hugging Face is a group that works with advanced data modeling neural networks. It has become a hub for people to upload their models and share their results with other people in the community. They are the creators of the transformers package that we will be using to develop our models that implement a BERT. This model will rely heavily on the prebuilt functions in the package to train the model and predict scores.

Since there are many different contexts in the dataset, it is essential to differentiate among the contexts because the similarity of the target and anchor can change drastically based on the context. One way of going about this is by combining the anchor and the context label to incorporate the context into the input. During this model, we chose to integrate context differently. Since the BERT model is pretrained and has an elementary understanding of English, we thought it could improve results by replacing the code for the context with some keywords involving the specific context for each observation. We were able to find a dataset on Kaggle that contained the title/keywords for each patent category. The dataset is shown below in figure XX. Instead of combining anchor and context from the training dataset, we merged the train data and the titles dataset. We created a new variable combining the anchor and the context title called input. This variable would be used to compare against the target to generate similarity scores.

The titles.csv can be found here: https://www.kaggle.com/datasets/xhlulu/cpc-codes (https://www.kaggle.com/datasets/xhlulu/cpc-codes)

	code	title	section	class	subclass	group	main_group
0	A	HUMAN NECESSITIES	A	NaN	NaN	NaN	NaN
1	A01	AGRICULTURE; FORESTRY; ANIMAL HUSBANDRY; HUNTI...	A	1.0	NaN	NaN	NaN
2	A01B	SOIL WORKING IN AGRICULTURE OR FORESTRY; PARTS...	A	1.0	B	NaN	NaN
3	A01B1/00	Hand tools (edge trimmers for lawns A01G3/06 ...	A	1.0	B	1.0	0.0
4	A01B1/02	Spades; Shovels {(hand-operated dredgers E02F3...	A	1.0	B	1.0	2.0
5	A01B1/022	{Collapsible; extensible; combinations with ot...	A	1.0	B	1.0	22.0
6	A01B1/024	{Foot protectors attached to the blade}	A	1.0	B	1.0	24.0
7	A01B1/026	{with auxiliary handles for facilitating lifting}	A	1.0	B	1.0	26.0
8	A01B1/028	{with ground abutment shoes or earth anchors f...	A	1.0	B	1.0	28.0
9	A01B1/04	with teeth	A	1.0	B	1.0	4.0

Figure 23: Titles file Format

The following model is built and trained through prebuilt functions provided by Hugging Face through the transformers package. I will be using three key functions from Hugging Face to produce a model. These three functions are TrainingArguments(), Trainer(), and predict(). These three functions are specially designed to work with any model loaded from Hugging Face. These functions will most likely work with other models created by yourself, but they work the best with models loaded from hugging Face.

After the Bert model is loaded using the from_pretrained() function and the dataset passes through the tokenizer to create the necessary embedding tables for the model, a TrainingArguments object is designed to change the model's parameters to optimize it. The TrainingArguments has 86 different parameters; we only use 11 of the parameters for my model. We used this function to specify the number of epochs, train batch size, eval batch size, learning rate, weight decay, and metric for interpreting the best model. We could have also changed the beta1, beta2, and epsilon of the AdamW optimizer, the default optimizer used for the function. Of course, we also could change the optimizer if we wanted to.

Function used to set up training arguments for hugging face trainer:

```
args = TrainingArguments(
    output_dir=f"/tmp/uspppm",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=Bert_Param.learning_rate,
    per_device_train_batch_size=Bert_Param.batch_size,
    per_device_eval_batch_size=Bert_Param.batch_size,
    num_train_epochs=Bert_Param.epochs,
    weight_decay=Bert_Param.weight_decay,
    metric_for_best_model="pearson",
    load_best_model_at_end=True,
    do_train=True
)
```

After creating the training arguments object, we made the trainer object that takes the model, training arguments, training dataset, validation dataset, tokenizer, and the metric function to evaluate the models. Running the train function on the train object creates the output below. Once the model finished training, it would run the predict function over the train set and the test set made from the training file.

Function used to train model (trainer):

```
trainer = Trainer(
    model,
    args,
    train_dataset=tr_dataset,
    eval_dataset=va_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```

Function used to compute pearson correlation coefficent for entire set:

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = predictions.reshape(len(predictions))
    return {
        'pearson': np.corrcoef(predictions, labels)[0][1]
    }
```

For the model, we implemented a function that would split the training dataset into folds for stratified cross-validation to measure the model's accuracy over multiple folds. We then ran the model over each fold, so one fold would be the validation set, and the other folds would be the training set. Since the distribution of scores isn't equal, we decided to use stratified sampling to ensure that the scores' distribution is similar among the train set and test set and each fold. The model would then run for each fold, with the validation set changing each iteration.

Epoch	Training Loss	Validation Loss	Pearson
1	0.006800	0.004880	0.975222
2	0.004800	0.005220	0.975149
3	0.003100	0.003067	0.980848
4	0.002000	0.003226	0.981543
5	0.001200	0.003128	0.982381

Figure 24: Trainer.train() output

Fine Tuning

With this model, we then ran multiple times, changing the parameters in an effort to find the optimal parameters for the model. Instead of running the model hundreds of times, we split the fine-tuning task into two parts. The first part was having standard learning rates and weight decay values, and the number of folds, epochs, and batch size would change. The results from this are found below in figure 18. The three accuracy scores marked with stars are the parameters that performed the best. These three sets of parameters would then be moved to the next fine tuning phase.

learning_rate	weight_decay	num_folds	epochs	batch_size	runtime	preds	train_accuracy
0.00002	0.005	3	3	16	0:13:56	0.882	0.737
0.00002	0.005	3	3	32	0:09:00	0.873	0.730
0.00002	0.005	3	5	16	0:22:55	0.906	*** 0.790
0.00002	0.005	3	5	32	0:14:50	0.892	0.760
0.00002	0.005	3	2	16	0:09:29	0.859	0.702
0.00002	0.005	3	2	32	0:06:06	0.830	0.659
0.00002	0.005	4	3	16	0:19:51	0.898	0.776
0.00002	0.005	4	3	32	0:13:04	0.898	*** 0.777
0.00002	0.005	4	5	16	0:32:39	0.920	*** 0.828
0.00002	0.005	4	5	32	0:21:12	0.898	0.776
0.00002	0.005	4	2	16	0:13:21	0.880	0.751
0.00002	0.005	4	2	32	0:08:40	0.859	0.702
0.00002	0.005	2	3	16	0:08:03	0.834	0.678
0.00002	0.005	2	3	32	0:05:09	0.827	0.650
0.00002	0.005	2	5	16	0:13:06	0.858	0.718
0.00002	0.005	2	5	32	0:08:23	0.846	0.686

learning_rate	weight_decay	num_folds	epochs	batch_size	runtime	preds	train_accuracy
0.00002	0.005	2	2	16	0:05:34	0.804	0.618
0.00002	0.005	2	2	32	0:03:33	0.778	0.600

Figure 25: Paramater Tuning Table 1

The next phase of fine-tuning was then used three sets of parameters stated above and find the optimal parameters of the optimizer, learning rate, and weight decay. The results from this fine-tuning are seen in figure 19. Three stars mark the parameters that performed the best and these are the final parameters used for our final model (learning rate = 0.00005, weight decay = 0.020, number of folds = 4, number of epochs = 5, batch size = 16). This set of parameters gave us a train accuracy score of 0.853 and a test accuracy score of 0.716

learning_rate	weight_decay	num_folds	epochs	batch_size	runtime	preds	train_accuracy
0.00002	0.005	4	3	32	0:13:04	0.898	0.777
0.00002	0.010	4	3	32	0:13:00	0.899	0.769
0.00002	0.020	4	3	32	0:12:56	0.889	0.765
0.00005	0.005	4	3	32	0:12:51	0.914	0.804
0.00005	0.010	4	3	32	0:12:51	0.914	0.806
0.00005	0.020	4	3	32	0:12:52	0.914	0.805
0.00010	0.005	4	3	32	0:12:52	0.911	0.806
0.00010	0.010	4	3	32	0:12:52	0.911	0.807
0.00010	0.020	4	3	32	0:12:59	0.913	0.809
0.00002	0.005	3	5	16	0:22:55	0.906	0.790
0.00002	0.010	3	5	16	0:22:47	0.906	0.789
0.00002	0.020	3	5	16	0:22:43	0.899	0.793
0.00005	0.005	3	5	16	0:22:40	0.912	0.817
0.00005	0.010	3	5	16	0:22:39	0.910	0.817
0.00005	0.020	3	5	16	0:22:39	0.909	0.813
0.00010	0.005	3	5	16	0:22:40	0.890	0.789
0.00010	0.010	3	5	16	0:22:43	0.892	0.794
0.00010	0.020	3	5	16	0:22:40	0.893	0.792
0.00002	0.005	4	5	16	0:32:39	0.920	0.828
0.00002	0.010	4	5	16	0:32:20	0.928	0.836
0.00002	0.020	4	5	16	0:32:23	0.924	0.841
0.00005	0.005	4	5	16	0:32:27	0.930	0.852

learning_rate	weight_decay	num_folds	epochs	batch_size	runtime	preds	train_accuracy
0.00005	0.010	4	5	16	0:32:24	0.927	0.851
0.00005	0.020	4	5	16	0:32:26	0.930	*** 0.853
0.00010	0.005	4	5	16	0:32:23	0.919	0.834
0.00010	0.010	4	5	16	0:32:24	0.917	0.832
0.00010	0.020	4	5	16	0:32:23	0.916	0.829

Figure 26: Paramater Tuning Table 2

The predict function produced a score for each data point. Since the scoring system for this project is not a continuous variable, we have to convert the prediction output to a discrete binned variable. To do this, we rounded each score to the closest label according to the prediction. Since there is a measurable difference between each label, we can easily turn the continuous variable into a discrete one. We then counted the number of points labeled correctly to obtain the model's accuracy. Figure 21 shows you the results of this process on the generated test set.

When using the optimal parameters with our model, we ran the model with the test file provided to us. We then produced a submission file with the format shown in figure 1, shown previously.

id	anchor	target	context	score	code	title	section	class	input	preds	label	Correct
123	water ..	varia..	B63	0.25	B63	SHIPS OR ..	B	63	title+anchor	0.264	0.25	TRUE
124	pipe b..	flute..	A01	0.0	A01	AGRICULT ..	A	1	title+anchor	0.272	0.25	FALSE
125	batter..	batte..	G01	0.5	G01	MEASURIN ..	G	1	title+anchor	0.795	0.75	FALSE
…	…	…	…	…	…	…	…	…	…	…	…	…

Figure 27: Test Split Results Format

Results: BERT Model

The tutorial method was trained with and without the context column. Both models achieved a Pearson R value of 0.72 with negligible difference between the two. Overall scores did vary from the two methods seen in the output file. Compared to our own implementation of the BERT model, the model trained without the context column concatenated into the strings performed more like ours. We do not have enough information to tell which way is the proper way as we do not know the actual scores for the test dataset.

Simple Models

In addition to the pre-made models discussed previously, simple model architectures were used based on what was learned in OR 610. The two main models attempted were based on homework 5: the CNN and the RNN. Additionally, even extremely simple models consisting of a linear layer directly after an embedding layer were attempted with no success. Within these models numerous sub models were also attempted. Unfortunately, none of these yielded tangible results, indicating the complexity of this problem.

For all models an accuracy of about 20-30% was observed which based on the number classes (5) indicates the model was doing little more than guessing. Loss values proved to be unresponsive to countless changes hovering at around 1.6 regardless of loss function or optimizer used.

Pre-model Code

All models attempted use the pre-trained GloVe embedding on a Pytorch embedding layer. This was done to add a layer of comparison, because outside of the context, the anchor and target need to both be similar for the overall row to have a high similarity score. (Garg, 2021)

```
#This pulls the information from the text file and creates lists that wi
vocab,embeddings = [],[]
with open('glove.6B.50d.txt','rt') as fi:
    full_content = fi.read().strip().split('\n')
for i in range(len(full_content)):
    i_word = full_content[i].split(' ')[0]
    i_embeddings = [float(val) for val in full_content[i].split(' ')[1:]
    vocab.append(i_word)
    embeddings.append(i_embeddings)

#Transforms
vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)

#insert '<pad>' and '<unk>' tokens at start of vocab_npa.
vocab_npa = np.insert(vocab_npa, 0, '<pad>')
vocab_npa = np.insert(vocab_npa, 1, '<unk>')
print(vocab_npa[:10])

pad_emb_npa = np.zeros((1,embs_npa.shape[1]))   #embedding for '<pad>' t
unk_emb_npa = np.mean(embs_npa,axis=0,keepdims=True)    #embedding for '

#insert embeddings for pad and unk tokens at top of embs_npa.
embs_npa = np.vstack((pad_emb_npa,unk_emb_npa,embs_npa))
print(embs_npa.shape)

#This trains the embedding layer
my_embedding_layer = torch.nn.Embedding.from_pretrained(torch.from_numpy

assert my_embedding_layer.weight.shape == embs_npa.shape
print(my_embedding_layer.weight.shape)
```

```
['<pad>' '<unk>' 'the' ',' '.' 'of' 'to' 'and' 'in' 'a']
(400002, 50)
torch.Size([400002, 50])
```

Figure 28: Code block for utilizing the pre-trained GloVe embeddings.

In addition to this a context embedding layer was created based on a list of all possible context labels. With these two embedding layers complete additional pre-processing was also done.

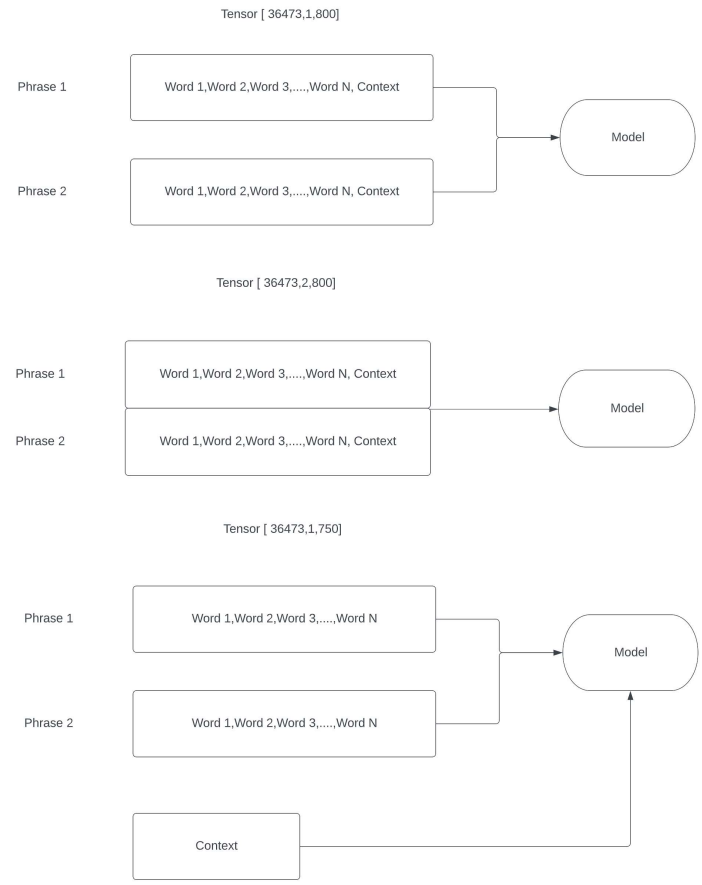
Since the data was read from a csv file using a pandas function it was in the form of a data frame with all each column containing a single string per row. To make this into a form useable by both the embedding layers and later models several transformations were performed:

- Each “phrase” for the anchor and target was split into individual “words”
- There words were then indexed based on the GloVe vocabulary
    - A check was performed to determine if the words in the phrase were in the vocabulary if not they were replaced with “<unk>”
- The phrases were then padded so they were the same length and passed through the embedding layer either before the model or in the model

```
def phr2wrd(column):
    for i in tqdm(range(0,len(column))):
        column[i] = column[i].split(" ")
    return column

def vocab_chk(column):
        for i in tqdm(range(0,len(column))):
            for j in range(0,len(column[i])):
                if column[i][j] in vocab:
                    pass
                else:
                    column[i][j] = "<unk>"
        return column

word2id = {word: ind for ind, word in enumerate(vocab_npa)}
id2word = {ind: word for ind, word in enumerate(vocab_npa)}

def sentence2emb(column, max_len):
    names_ix = np.zeros([len(column), max_len], np.int64) + word2id['<pa
    for i in tqdm(range(0,len(column))):
        name_ix = list(map(word2id.get, column[i]))
        names_ix[i,:len(name_ix)] = name_ix
        #print(names_ix[i])
    return names_ix

cont_vocab = pd.read_csv('Context_List.csv')
contexts = list(cont_vocab.context)
contexts
cont2id = {cont: ind for ind, cont in enumerate(contexts)}
id2cont = {ind: cont for ind, cont in enumerate(contexts)}

def cont2emb(column):
    cont_ix = np.zeros([len(column), 1], np.int64)
    for i in range(0,len(column)):
        cont_ix[i] = cont2id[column[i]]
    return cont_ix

context_vocab = cont2emb(contexts)
context_vocab2 = torch.LongTensor([context_vocab])
context_emb_layer = nn.Embedding(136,50)
```

Figure 29: Code block for pre-processing

Finally, the tensors were manipulated in numerous ways in an attempt to get a useable output, some of the attempts can be seen below:



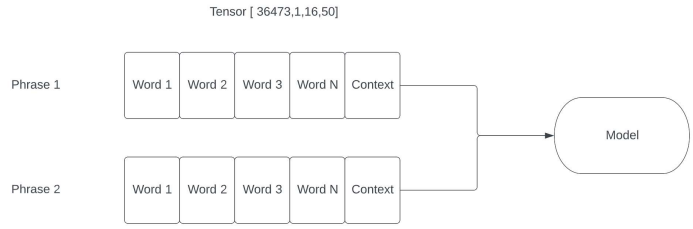


Figure 30: Attempts of manipulation of tensors

Models

The goal of these models was to see if you could get a useable output with very simple models. The main models attempted were CNN based, using 1 dimensional or 2 dimensional depending on the input tensor dimensions with a majority of implementation being based on homework 5. The reasoning being that the features captured in images and converted to tensors are not much different than features captured in phrases and converted to tensors so the hope is the model can find some connection based on these features. Some architectures implemented range from a single convolution layer with a linear layer to follow, to more complicated multi-layer and multi-channel models like the one below (Brownlee, 2018):

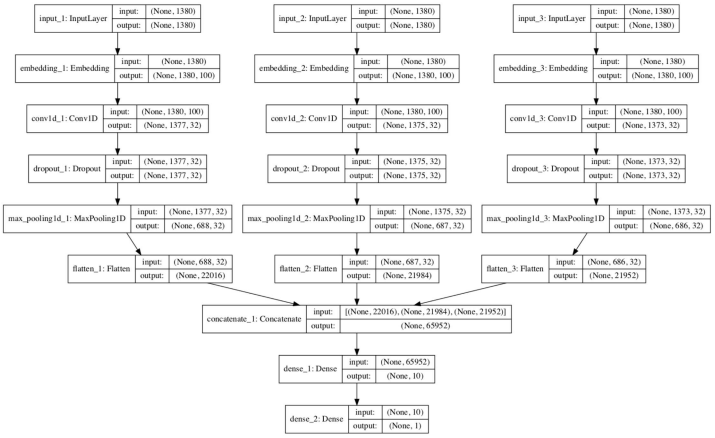


Figure 31: Plot of the multichannel CNN for text

As with previous components, various loss and optimizer functions were attempted like NLLLoss, CrossEntropy and even MSE as well as Adam and SGD and others.

The models would generate outputs consistently but through various training runs and countless epochs, no improvements were noted in loss.

Problems

The exact cause of the problem was unfortunately not discovered but the main hypothesis has something to do with the recording of gradients and implementation of Python/PyTorch on the new Apple M1 architecture. A consistent problem this semester was the use of PyTorch and CUDA on the M1 Pro chip which is basically using an emulator to run python since the native implementation is not available yet.

Also, in certain trials the following error message would appear:

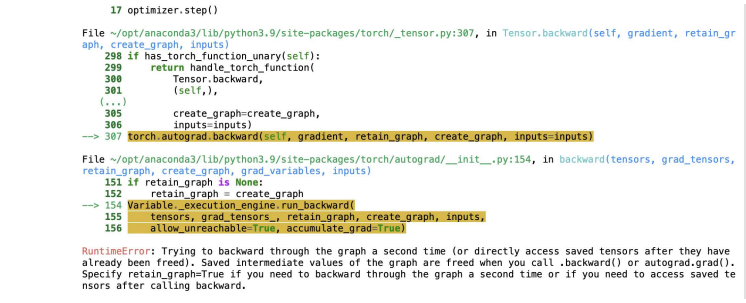


Figure 32: Error message from certain trials

This message indicates that code is trying to backward through twice, regardless of including optimizer.zero_grad() or even the argument retain_graph=True, this message would persist. This error could be due to how the train function, dataset or dataloader were implemented but numerous tests yields no change.

Conclusion, Future Work, and Limitations

In conclusion, we created four different models and three produced predictions. Of the three models that produce predictions, the Hugging FACE function BERT model produced a 72% accuracy on a test set split from the initial training dataset. In contrast, the MaLSTM produced an accuracy of 40%. As for the true accuracy of the models, it is hard to determine which model performs the best due to the project's materials design. The test file provided doesn't give you scores, making it challenging to assess accuracy. In addition, the competition has its own test set not given to the public to evaluate accuracy to determine the competition's winners. If we wanted to continue work on this project, we would attempt to get the last model working and try different models on Hugging FACE other than the BERT base case.

References

[1] https://huggingface.co/docs/transformers/index (https://huggingface.co/docs/transformers/index)

[2] Barla, Nilesh. (March 21, 2022). How to Code BERT Using PyTorch – Tutorial With Examples. Neptune Blog https://neptune.ai/blog/how-to-code-bert-using-pytorch-tutorial (https://neptune.ai/blog/how-to-code-bert-using-pytorch-tutorial)

[3] Denaux, Ronald. (Nov 19, 2019). Semantic Claim Search

[4] https://huggingface.co/docs/transformers/main_classes/trainer (https://huggingface.co/docs/transformers/main_classes/trainer)

[5] https://huggingface.co/docs/transformers/v4.19.0/en/main_classes/trainer#transformers.TrainingArguments (https://huggingface.co/docs/transformers/v4.19.0/en/main_classes/trainer#transformers.TrainingArguments)

[6] Denaux, Ronald, et. al., Semantic Claim Search, (2019), Github Repository. https://github.com/hybridnlp/tutorial/blob/master/07a_semantic_claim_search.ipynb (https://github.com/hybridnlp/tutorial/blob/master/07a_semantic_claim_search.ipynb)

[7] Devlin, Jacob; Chang, Ming-Wei; Lee, Kenton; Toutanova, Kristina. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/arXiv.1810.04805 (https://doi.org/10.48550/arXiv.1810.04805)

[8] Song, Xinying. (2021). A Fast WordPiece Tokenization System. https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html (https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html)

[9] Brownlee, J. (2018, January 12). Hpw to Develop a Multichannel CNN Model for Text Classification. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/develop-n-gram-multichannel-convolutional-neural-network-sentiment-analysis/ (https://machinelearningmastery.com/develop-n-gram-multichannel-convolutional-neural-network-sentiment-analysis/)

[10] Garg, T. (2021, April 25). Load pre-trained GloVe embeddings in torch.nn.Embedding layer… in under 2 minutes! Retrieved from medium: https://medium.com/mlearning-ai/load-pre-trained-glove-embeddings-in-torch-nn-embedding-layer-in-under-2-minutes-f5af8f57416a (https://medium.com/mlearning-ai/load-pre-trained-glove-embeddings-in-torch-nn-embedding-layer-in-under-2-minutes-f5af8f57416a)