Imitating Commonsense Reasoning using Natural Language Processing

William Gibson
Undergraduate Computer Science Student
Northwest Nazarene University
Email: wgibson@nnu.edu

Jack Rocco
Undergraduate Computer Science Student
Northwest Nazarene University
Email: jrocco@nnu.edu

***Abstract*—This project seeks to train an AI to be able to accurately emulate the common sense reasoning found in humans. Through the use of Natural Language Processing and Deep Learning, this AI will be trained and tested on a dataset of thousands of sentences that represent common sense reasoning, either correctly or incorrectly. We have found that our method can produce correct results 65 percent of the time, which is quite a significant find, and is evidence that a more well developed model could, in theory, detect whether a statement follows common sense logic with near perfect accuracy.**

***Index Terms*—common sense, natural language processing**

I. Introduction

Commonsense reasoning was chosen for this project due to the challenges that AI faces in predicting outcomes based on limited or ambiguous datasets. AI must be capable of replicating folk psychology, the ability to reason through behavior and intentions, and naive physics, the ability to understand the physical world. The project uses a preset dataset of correct and incorrect statements. With this dataset, the AI can predict whether a statement is true or false based on commonsense reasoning. The application of common sense reasoning can allow for the improved implementation of humanized responses.

II. Background

AI common sense reasoning enables the use of incomplete or ambiguous data to make decisions based on inferred probabilities. The objective of this project is to predict the validity of a statement using common sense reasoning. The dataset used in the project was collected by Hassam Asif and obtained from Kaggle. To embed the gathered data, we utilized SBERT, a pre-built sentence transformer. Next, the sklearn library was used to build a Logistical Regression model, where the encoded sentences and labels were fed to create the training samples. The following sections will summarize our approach to common sense reasoning prediction.

III. Data

A. Description of Data

The dataset we used was the Common Sense Evaluation Data dataset, by Hassam Asif. This dataset includes a number of different files, which all include sentences related to Common Sense reasoning. Each of the files has the same 7 columns, a sample of which are shown below. For the purposes of getting the model running, we used the test.csv file, which has 1000 entries for each column. Once we had a working model we also ran the train.csv file, which has 10,000 entries for each column. The Confusing Reason2 and Right Reason2 columns share the same statements because of their common sense logic, but they fail to follow a humanized structure. Fig. 1 shows a sample of the dataset.

Correct Statement	Incorrect Statement	Right Reason1	Confusing Reason1	Confusing Reason2	Right Reason2	Right Reason3
when it rains humidity forms	when it is hot humidity forms	hotness will evaporate water	Humidity is a measure of moisture in the atmosphere.	Laundry will not be dry because of the humidity.	Laundry will not be dry because of the humidity.	Water makes humidity, not temperature.
Grizzly bears love honey.	Grizzly bears hate honey.	Honey is good for grizzly bear's growth	Grizzly bears have been observed taking stings from swarms of bees to access honey.	Bees cannot eat grizzly bears	Bees cannot eat grizzly bears	Grizzly bears have been observed in the wild seeking out honey to eat
shrimps can live underwater	dogs can live underwater	dogs haven't organs to breath underwater	dogs can swim in the water	Some dogs hate the water	Some dogs hate the water	Dogs need air to breath, so they cannot live underwater

Fig. 1. Example of Data.

B. Preprocessing

The entire file was imported directly into a Pandas dataframe, to allow for simple manipulation of the data, and to give us a reasonably way to view the date. As we moved into the preprocessing and training phases we cut the number of columns we were using to just 2. We were looking to do a binary classification, and so we cut it down simply to the Correct Statement and Incorrect Statement columns. Once we had this ready to go, we moved forward with defining and training our model.

IV. Model and Approach

A. Defining our Model

At the beginning of our journey into developing this model, we started with the bert-base-uncased model. This model was used for our sentence embeddings, and was then put together with a pooling model into the full model to be used for fine tuning. We went with fine tuning the model for our specific dataset, in order to make sure our model was more appropriate for our dataset. We were hoping to end up with a model specifically fit for our dataset and problem. Fig. 2 shows how we defined this model.

```
# Define the model
model_name = 'bert-base-uncased'
word_embedding_model = models.Transformer(model_name)
pooling_model = models.Pooling(word_embedding_model.get_word_embedding_dimension())
model = SentenceTransformer(modules=[word_embedding_model, pooling_model])
```

Fig. 2. Defining the Model

B. Format Data

To accomplish this, we needed to take our data and get it into a format that the model would accept. We started by taking our set of correct statements and incorrect statements, currently in a 2d array, and turning it into InputExamples. These are a datatype in the Sentence Transformer library that takes a text and a label and puts them together. These can be used along with the models in the Sentence Transformers library to make sure everything is in a common format. Fig. 3 shows how we created these input examples.

```
for entry in sentences:
  example = InputExample(texts=[sentences[num][1]], label=sentences[num][0])
  all_examples.append(example)
  num += 1
```

Fig. 3. Create InputExamples

C. Loss Function and Dataloader

Once we had our data in the correct format, we had to wrap it up in a dataloader to import it into the model. The dataloader does all of the heavy lifting in terms of getting the data in an iterable mode. Now that each individual example is ready to go, we need to put it together into a complete package for the model to accept. Once this is good to go, we also want to get a loss function setup. We used SoftmaxLoss, again from the Sentence Transformers dataset, because it seemed to perfectly fit our problem. Once everything was safely packaged up, we could import everything into our model. Fig. 4 shows the code for these sections.

```
# Define the loss function and data loader
train_loss = losses.SoftmaxLoss(model=model, sentence_embedding_dimension=model.get_sentence_embedding_dimension(), num_labels=2)
train_examples = [InputExample(texts=[s[1]], label=s[0]) for s in sentences]
train_dataloader = DataLoader(train_examples, shuffle=True, batch_size=32)
```

Fig. 4. Loss Function and Dataloader

D. Fine Tuning the Model

Model.fit is a TensorFlow based call that puts everything together. This single line does it all. It reads everything from our dataloader and trains our deep learning model on the information we have provided. This call does a lot, and in our implementation, that ended up being more of a curse than a blessing. Because the call was relying on and calling to a number of different things, I would end up with errors that stretched 11 calls deep in the call-stack. This made creating/debugging the model nigh on impossible, right up until the wonderful Enoch Levandovsky came to our rescue, and gave us a different approach. Fig. 5 shows the code.

```
# Train the model
num_epochs = 5
warmup_steps = int(len(train_dataset) * num_epochs * 0.1)  # 10% of train data for warm-up
model.fit(train_objectives=[(train_dataloader, train_loss)],
          epochs=num_epochs,
          warmup_steps=warmup_steps)
```

Fig. 5. Finetuning Code

E. A New Approach

The approach that Professor Levandovsky ended up using was a bit different, and didn't require any of the clunky dataloaders or InputExamples. Enoch gave our data another row of labels which were the opposite of the original labels (if it was a 0, the new label was 0, 1 and vice versa). This essentially did one hot encoding on our labels, which seemed to be more in line with what Sentence Transformers was expecting. We used a different model, which was 'all-mpnet-base-v2'. This model was used to create embeddings for each of our sentences. Fig. 6 shows the embeddings line.

```
embeddings= model.encode(np.array(sentences)[:,2])
```

Fig. 6. Create Embeddings

F. Training the Model

We then put our encoded sentences and our labels together in a LogisticalRegression model from the sklearn library. This used a 90 percent train to 10 percent test ratio, giving us 1800 training samples in our first dataset, and a whopping 18,000 training samples in our second dataset. Once all of this was run, our evaluation would finish off the program, and give us the final say on accuracy. Fig. 7 shows the last bit of code to finish off the training portion.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X=embeddings
y =  np.array(sentences)[:,0]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
clf = LogisticRegression(random_state=0).fit(X_train, y_train)

clf.score(X_test, y_test)
```

Fig. 7. Training the model.

G. Note:

It is important to note that this version of our program did not fit our model to the training data, so the final evaluation was a bit skewed from what it could have been if our original approach had worked. Needless to say though, we are still proud of the results, even as they are.

V. Evaluation

A. Original Evaluation Model

When it came to evaluation our original plan while we were still using our BERT based model and the Sentence Transformers dataset was to use the BinaryClassificationEvaluator available from Sentence Transformers. This evaluator is built specifically around being able to use it to evaluate a models

ability to perform biniary classification. Since we had cut our dataset down to just the two classes, Correct Statement (0) and Incorrect Statement (1), we were in a perfect position to use this. While our program never ran far enough to get to this line of code, we are still fully confident that it would have worked as intended.

B. New Evaluation Model

The evaluation that ended up happening instead was the simple clf.score call. 'clf' was simply the name we associated with our LogisticRegression model, and the .score is a method built into said model which outputs the accuracy of the model based on the testing data you import. Our results are as follows:

C. Results

The smaller set of data, trained on 1800 samples and tested on the remaining 200 had an accuracy of 53 percent. This is a bit underwhelming, as that is barely higher than randomly guessing, but 3 percent was a surprising enough margin to be "succeeding" by when it came down to all the problems we had experienced throughout the project.

The larger set of data, trained on 18,000 samples and tested on the remaining 2000 had an accuracy of 65 percent, which was a much better show of the models capabilities.

VI. Implications

The implications of this project are tremendous in my opinion. The proof, no matter how subtle, that AI can find a difference between common sense statements that are true and those that are false is astounding, especially for a model based purely on the logic side of the spectrum behind the scenes. It can fake the "logical reasoning" of a human, and though it might not be to a convincing degree, it is an impressive feat nonetheless. While doing research for this project we came across a GitHub project that was very similar to what our original vision had been, with it predicting between a number of different classes. This project had an accuracy above 90 percent for its dataset, which shows that if we had been able to successfully fit the model to our dataset, we could have gotten a much higher accuracy.

VII. Conclusion

Common sense reasoning is a relatively difficult thing for a computer with no sense of human reasoning to pull off. But, a computer trained on thousands of sentences will be able to fake it decently well. This project in particular has shown off an incredibly simple, baseline approach to this topic. Building off of this with further developments, such as specifically fine tuning an AI model to detect common sense, as well as giving it the ability to more deeply train on this data, would vastly improve its ability to logically find a way to emulate human reasoning and intuition. And that, is what Artificial Intelligence is all about.

References

[1] https://www.kaggle.com/datasets/hassam361/common-sense-evaluation-data.
[2] https://www.sbert.net/docs/pretrainedmodels.html
[3] https://scikit-learn.org/stable/modules/generated/sklearn.modelselection.traintestsplit.htmlMagnetism
[4] https://github.com/UKPLab/sentence-transformers/blob/master/examples/training/other/trainingbatchhardtrec.py