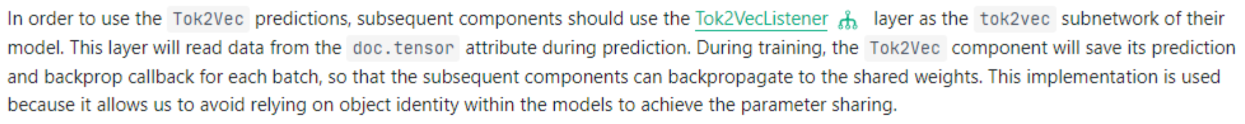
Features

There were several features that were designed throughout the design cycle, broken down into these main categories:

- BLEU Scores
    - Algorithm was discussed in class, taking the product of the proportion of matching n-grams times a brevity penalty to get our score. The nltk library has a function for determining the BLEU score for any number of n-grams. In my final version, there were 4 features related to this, BLEU_1 through BLEU_4. The number in each feature represents the maximum number of N-grams that were being considered. Additional n-grams after this point seemed to not have much impact, and using less n-grams removed some accuracy from the model
- NIST Scores
    - A very slight modification to the BLEU algorithm, instead of using geometric means of the n-gram overlaps, uses arithmetic mean. Initially tested with n-grams up to 5, but in the end this feature was dropped. Comparatively, BLEU scores proved to give higher accuracy, and when both BLEU and NIST were used as features simultaneously, accuracy suffered. NIST was implemented from the nltk library as well
- METEOR Scores
    - Acronym for Metric for Evaluation of Translation with Explicit ORdering. Similar type of feature as BLEU, but a little more expansive. Inside of the meteor score are features like synonym matching and lemmatization. Later, it will be discussed how lemmatization was dropped as a preprocessing feature early on, but I believe that considering lemmatized sentences in one of our features allowed us to see different aspects of our sentences. METEOR was another module pulled from the nltk.translate library
- Cosine Similarity
    - For this feature, the spacy library was used. Each sentence in our input was vectorized, and then the cosine similarity between the sentences was computed. While this feature did have an improvement on the accuracy, ultimately I removed the feature due to concerns of the involvement of neural networks. The Spacy documentation makes mention that subnetworks are used in the tok2vec function that is a part of the generation of the word embeddings. It’s removal from the project had a very small impact on our dev set accuracy, so to err on the safe side of the project conditions, I removed it. An implementation could have been done with a BagOfWords implementation of a word vector, but from my readings it seemed to me that this would be unsatisfactory if there were words in the dev or test set that weren’t in the training set, which I viewed as likely.


In order to use the `Tok2Vec` predictions, subsequent components should use the Tok2VecListener layer as the `tok2vec` subnetwork of their model. This layer will read data from the `doc.tensor` attribute during prediction. During training, the `Tok2Vec` component will save its prediction and backprop callback for each batch, so that the subsequent components can backpropagate to the shared weights. This implementation is used because it allows us to avoid relying on object identity within the models to achieve the parameter sharing.

- Character Based Features
    - There were 4 overarching features related to the characters in each of the two sentences to compare. We focused on features with 1-grams (looking at chars individually) and 2-grams. These 4 features were:
        - Union of two sentence's character grams (mimicking combined sentence length)
        - Intersection of the character grams (How many in common)
        - Number of n-grams from sentence 1
        - Number of n-grams from sentence 2
    - It became apparent that character 1-gram features were unhelpful, as overlapping characters in each sentence really doesn't tell much about how similar they are. 2-grams did show some useful information, which I accredit to similar reasons as the METEOR score, comparing pairs of chars may have allowed some words that have similar root words to get positive correlations
- Word Based Features
    - Very similar to character-based features in design, with focus on single words or bigrams of words, and the same 4 features for each (word union, word intersection, number of words in sentence 1, number of words in sentence 2). Unlike the character-based features, the bigram-based features seemed to add little to our accuracy, and were too similar to the unigram features, so they just cluttered our model. There was a small use of the nltk library to extract the n-grams, but otherwise the code was simple and personally made

So, in my final model, there were 13 Features total:
- BLEU_1 - BLEU_4 (4 total features)
- Meteor Score
- Character Bigram Features (4 total features)
- Sentence Unigram Features (4 total features)

Data Preprocessing
When I first began making my model, before I started defining the features I started preprocessing on the train/dev data. I did some very standard NLP preprocessing:
- Removing stop words (words that add little meaning to text, like "the" or "a")
- Lemmatizing text (Removed the suffixes of words to just leave the root word)
- Removed punctuation (including quotation marks, periods, commas, etc.)
- Made all words lowercase

Whenever I got my model working, and was looking to make improvements, one of the papers that I read on this subject, which also informed some of the features that I tested, mentioned that they used only minimal preprocessing. It was claimed that removing too much information from the original sentences ran counter to our purposes. So, I made another preprocessing function that only made all of the words lowercase, and immediately my score was improved by a pretty large margin (~2% at the time it was first implemented). This ran very counter to my intuition, so I tested the old preprocessing function occasionally as new features were added or removed, but the minimal preprocessing worked better in every case I tested, so it is the version used for my final model's input sentences.

<u>Feature Preprocessing</u>

There were two feature preprocessing steps that were used in my models:

- Scaler function (standardize/normalize elements of each feature)
- Principal Component Analysis(to ignore features with little variance)

Both of these were implemented with the sklearn libraries. There were a few different scalers tested, but the StandardScaler() function seemed to work best for my purposes. The goal of this scaler is to normalize all of the elements of each feature, so we don't end up with features that have larger scores than others just by nature of design, which could throw off the weights that are assigned to them.

The Principal Component Analysis was another step, but the number of components that were kept was actually experimented on while determining the parameters of our best model. As stated, this was implemented in order to help remove the impact of some of the parameters with little variance. This was implemented with sklearn's decomposition library, and the only parameter to tune was the number of components to keep, which was tuned with the methods described below.

<u>Optimal Model Search</u>

By far the most helpful way to determine the optimal model for my features was the GridSearchCV function from sklearn. When I first started testing my features, I made ~8-9 models, one for each of the logistic regression penalties, and one for each of the Support Vector Classifier kernels. These simple model comparisons gave me the impression that the variations of logistic regression models tended to have a higher score than the SVC models in general, but there were many hyperparameters to tune, and copying the model over and over would be highly inefficient. The GridSearchCV function takes a dictionary of parameters, and tests each combination of inputs. This allowed me to test thousands of models very simply. Given the relative simplicity in the logistic regression, it was able to test these models very quickly, where the SVC model search took upwards of half an hour. Part of this model optimization was determining the number of components that were optimal as part of the PCA in our feature preprocessing. Given the time it took to find our optimal SVC models, I just took the best option that came out of it for one of the runs, and used that as my baseline comparison against the logistic models as I made changes to the features, as 30 minute processing times for small changes would have made the development process much longer. Once I settled on my final model, I did do a simplified search of the SVC models, leaving out parameters that didn't have much impact on accuracy previously, and still found the logistic regression models still performed better. In the end, my best model was a logistic regression model with a newton-cg solver (from my reading it is a form of a gradient descent model that uses second derivatives in its calculations) with no penalty in the algorithm, and 12 PCA components.

<u>Algorithms and Libraries</u>

<u>Nltk(main library for feature definition)</u>

- nltk.word_tokenize
- nltk.translate.bleu_score.sentence_bleu
- nltk.translate.meteor_score.single_meteor_score
- nltk.ngrams

Pandas(used to read and store data from input files and store feature values)

- pandas.read_csv
- pandas.DataFrame

Sklearn (Main library for creating and optimizing model)

- sklearn.linear_model.LogisticRegression
- sklearn.model_selection.GridSearchCV
- sklearn.pipeline.Pipeline
- sklearn.preprocessing.StandardScaler
- sklearn.preprocessing.MinMaxScaler

Csv

- No function used, but QUOTE_NONE value (literal value 3) used in pandas read_csv file to allow it to read in sentences not enclosed in quotes

Itertools

- Itertools.zip_longest
    - Used to loop through each sentence in both arrays of sentences simultaneously

Warnings

- warnings.filterwarnings
    - Used to suppress warnings from BLEU functions whenever a sentence pair had a zero-output due to no matching n-grams, which was our intended usage

Experience and lessons

This was a very interesting hands-on look into simple machine learning models. Some of the most interesting takeaways for me was that less can be more when it comes to features. I had the impression that throwing all the features in you could think of, and letting the model determine which were important through weight assignment would be the best way to approach, but it became obvious lots of low weight features were detrimental. Also, when I first started working on this, I was failing to read in a few hundred elements of my training data due to an improperly setup parsing function, and when I realized my mistake and allowed the extra data in, my model's accuracy was immediately improved. This was a clear and simple way to see how much of an impact extra data can have on the models. By far the largest portion of my time on this project was spent on determining what features would be best, and my final accuracy score (around 68-69% on the dev set) didn't meet some of even the baseline features in the models detailed in many of the papers that I read. This was very disheartening at first, but I think I see a little better now that you really cannot extrapolate feature results on one data set to another. Even though the task for all the papers was the same, many of them had differences that make the dataset we used not able to be directly compared (unbalanced output classes, shorter/longer sentences, more narrow/broad number of pairs to test on, etc). I am sure that there is room for improvement in my model's accuracy, and I am interested in testing my model on the datasets used in those papers, and comparing my model with my classmates to see what we did differently. Overall, this was a very interesting experience, and I hope that we will be able to get some feedback or recommendations before diving into our final project on improvements to design philosophy.