**Clustering for Chat Bot Intent Architecture**

Samuel Sears

Bellevue University

DSC 550: Data Mining

Professor Becky Deitenbeck

Aug. 5, 2020

**Clustering for Chat Bot Intent Architecture**

With companies transitioning to online services for customers in recent years, customer support has also been transitioning to online messaging as opposed to phones. For phone support, it is often difficult for massive companies to have representatives trained on all the different skills they get calls about. Therefore, there is a need to narrow down what the customer needs help with prior to connecting them with a support representative. This can be accomplished with an ever familiar IVR. Similarly, with online messaging, companies can staff chat queues based on where the customer is entering the chat system from. The goal here is to increase customer satisfaction by getting them to a person that can help them quickly, but also to reduce the staffing needs by reducing the transfer rates or long conversations stemming from the rep not knowing the answer to the customer’s question. However, there are now even more ways to reduce customer support staffing for companies by implementing chat bots that can help the customers with repetitive chats. This allows the company to use their skilled customer service representatives to handle questions too complicated for a programmed response instead of answering the same question over and over again throughout the day. That being said, the chat bot has a critical requirement of being able to determine the reason for the customer contact.

**Problem**

When a company decides to implement a chat bot, they typically will have a large dataset of call or chat transcripts to support a use case. They are hoping that a large portion of these contact could be handled by a programmed response from a chat bot. However, in order to get the customer to the correct programmed response, the bot must be able to accurately classify the user’s intent or question. Chat bots are able to do this by training a machine learning classification algorithm with groups of similar utterances, or customer inquiries. Each group of utterances would represent an intent which would have a singular programmed response. However, that response could be to disambiguate further on the needs of the customer. Therefore, the intents could be broader or narrower depending on the intent classification performance. The problem arises when trying to determine what the intents should be.

One way a company could determine intents would be to label the utterances in the dataset one by one with the intent they think the utterance should fall under. With a dataset typically consisting of thousands of utterances, this would be a very long and tedious process. Further, if a team is working on this project, there would be a high probability of team members classifying intents differently. For example, one team member might classify three utterances into one broad intent where another team member may classify those same utterances into three narrower intents. In the end, you may wind up with a model with a whole bunch of overlapping intents with varying degrees of breadth.

Another option would be to filter the utterances by keywords or phrases. For example, perhaps you think an intent for when the customer wants to delete their account would be appropriate. A problem you will run into with this would be if you filtered by ‘delete account’ you may get different utterances with completely different meanings. One utterance may be talking about deleting a user account and the other may be talking about deleting a bank account from their billing methods. Further, you may have trouble determining all the different key phrases that would identify the intent. If we could determine the intents just through identification of key phrases in a sentence, then there would not really be a point to having a fancy machine learning classification algorithm to do that same task.

A third option would be to deploy the intents iteratively. So, for a minimum viable product we may identify a few of the most common inquiries that come into the support center. However, what happens when one of those inquiries is to make a payment and there are other inquiries that come in with similar language patterns like update payment method. There will be a high chance that the bot will classify any payment intent as the one payment intent the bot is trained on mistakenly.

**Solution**

The solution for the issues arising from iterative deployments is to train the bot on the entire domain or not at all. Training the bot on parts of the domain at a time will have inherent issues with intent confusion. To solve the issue of time and user error in thinking of all the key words that denote the intent, we can cluster the available dataset using similarity between sentences as a distance measure. The resulting clusters would become the intents for the bot.

**Methods**

To test this solution, a customer support dataset was used to mimic what might be available to a company. Two clustering methods were used for creating the intents. Evaluation of the clustering methods was completed by training a Lex chat bot using the clusters as intents and running a validation dataset against it.

**Dataset**

The dataset used came from a Bitext, a company focused on creating chat bot intent architectures (Bitext, 2019). Their methods include not only clustering, but also producing largely synthetic intents. However, that is a whole topic on its own. The dataset includes over 20,000 utterances labeled with 27 unique intents in the customer support domain. For my own purposes, I did not use the intents Bitext provided in the dataset, but I did keep in mind that there were likely 27 distinct clusters.

**Hierarchical Clustering with Cosine Similarity**

My first attempt at clustering was to create a hierarchical clustering algorithm from scratch using cosine similarity as my distance measure. Each step in my algorithm compared each utterance to every other utterance or cluster and computed the cosine similarity. Cosine similarity was determined by removing special characters and stop words from the sentences and converting them to vectors (GeeksforGeeks, 2019). A high cosine similarity was determined when the cosine distance was low. The cosine distance was low when the sentences contained the same words. Whichever utterances/clusters were the most similar were combined in each step. Further, the distance between an utterance and a cluster was determined by taking the average of the similarities between that utterance and each utterance in the cluster. The steps were completed until there were no combinations with a similarity of a certain threshold. Different thresholds were tested, but I stopped at a threshold of 0.2. This seemed to give clusters of equal length and relatively similar utterances.

To evaluate how the clustering algorithm was performing I tracked the total time taken, the number of utterances, the average within cluster similarity, each cluster’s within cluster similarity, the utterances in each cluster, and the cluster size. The overall within cluster average similarity for a threshold of 0.2 was 0.38 with the best cluster being 0.59. This is on a scale of 0 to 1. The problems I found with this method was that it only considered lexical similarity and not semantic similarity. In other words, if two sentences had the same words, they would be considered similar, regardless of the meaning. An example of a cluster that expressed this issue was cluster 19 where one sentence was asking for a copy of a bill and the other for reimbursement. The two sentences were deemed similar simply because they had the word obtain in them. Beyond the issues with semantic meaning, the algorithm simply took too much time. For less than 100 utterances it took the algorithm almost 30 seconds. This was not scalable to the entire dataset.

**Hierarchical Clustering with Word2Vec Word Embedding**

There were two issues with the first attempt; the time it took to cluster the dataset and the lack of consideration for semantic meaning. The first issue was solved by utilizing the AgglomertiveClustering algorithm from SciKit-Learn (SciKit-Learn, 2020). The second issue was solved by first training a Word2Vec word embedding model from the Gensim library (Brownlee, 2017).

The first step for this method was to load the data from the Bitext file complete the word embedding. This was done by first preprocessing the utterances which tokenized the words of the sentences, removing stop words and special characters. The result of preprocessing the utterances was a list of lists where each list was a list of words corresponding to the utterance. The word embedding model was trained by passing the preprocessed sentences to the Word2Vec method. The Word2Vec model outputs a vocabulary where each word has a 50-dimensional corresponding vector. Two words that appear close to one another and frequently in the same utterance would be given similar vectors where if you took the Euclidean distance between them it would be small compared to two words that were not similar.

The next step was to get the utterances into a format that the clustering algorithm could use. To accomplish that, each unprocessed sentence was again processed and the Word2Vec model’s vector for each word in the sentence was obtained. The mean was then taken for each word vector in the utterance as a representative vector for that utterance. The output of this step was a list of vectors corresponding to each utterance in the dataset.

Finally, the utterance vectors were passed to the AgglomerativeClustering method from SciKit-Learn. Different parameters were investigated, but there were only three final ones that gave the best result. For number of clusters, 27 was chosen since it was known that the Bitext dataset had 27 clusters. Next, Euclidean distance was used for the distance measure. This was used over cosine similarity because the magnitude, not just the angle between the vectors, was important. Finally, the linkage criterion used was average, meaning the clustering algorithm looked at average distance between the merging clusters as opposed to other options like reducing the amount of variance in clusters (SciKit-Learn, 2020). The result gave 27 clusters, but 4 of them had 5 or less values. The size of other clusters ranged from 103 to 3725 utterances. After briefly looking through the utterances in each cluster, it was determined the results were promising enough to move on to testing in Lex.

**Testing**

As stated before, the cluster method was tested by using the clusters as intents in an Amazon Lex chat bot. Since AWS Lex only allowed for a maximum of 1500 utterances per intent, the split between testing and training data was predetermined. The goal behind splitting the training and testing dataset is to maximize the training data while still leaving enough testing data to be reliable. Training data was maximized by splitting the dataset 60% test data, 40% training data. Since the test data was comprised of over half of the entire dataset, the results of testing were determined to be more than adequately representational. Further, the clusters with 5 or fewer utterances were placed in the testing dataset as unhandled utterances since there was not enough data to adequately train and test on those. Randomly splitting the data was handled using SciKit-Learn’s train\_test\_split method. The two sets were then exported into Excel for the next step.

The next step of testing involved importing the training dataset into Lex. This was a tedious process as Lex has a strict json format used for bot, intent, and slot importing and exporting. In order to import the dataset, I used formulas in Excel to get the utterances in the correct format for copying and pasting into a blank bot’s json export. I then only had to re-import that bot and the intents were loaded into the Lex console. I experimented with exporting the intents from my notebook to the json format but was unsuccessful.

Finally, the bot was built and published in the Lex console. With that completed, I was able to run all of the withheld testing data through the bot and record the predicted intent.

**Results**

In the end, 8332 utterances were used to train the Lex bot’s 24 intents. The number of utterances per intent ranged from 7 to 1457. Comparatively, 12,390 utterances were tested against the model with proportions of utterances per intent tested equivalent to that of the training set.

**Performance Metrics**

For each intent, the number of actual utterances, predicted utterances, and correctly predicted utterances were compiled. Overall, there were 12,089 correct predictions, where the predicted intent matched the actual intent, out of the 12,390 utterances tested. Therefore, the accuracy for the entire test was 98%. Further, the recall, precision, and F1 score were also calculated for each intent. The recall for each intent showed the proportion of tested utterances for that intent which were predicted correctly. In other words, what percentage of the utterances tested did the bot catch. Recall scores ranged from 89% to 100%. Precision, on the other hand, showed how precise the bot was when it did predict an intent. In other words, when the bot predicted a specific intent, how often was it correct. Precision is typically correlated with a better user experience since if a user gets to an incorrect intent, it is a worse experience than if the utterance were to just be unrecognized. Luckily, the precision scores for the different intents ranged from 93% to 100%. Finally, the F1 score, or harmonic mean between recall and precision, was also calculated for each intent. The F1 scores ranged from 93% to 100%. Therefore, overall and intent to intent, the bot performed extremely well with the intents determined by the clustering algorithm.

**Confusion Matrix**

The last tool used to visualize the performance of the bot was a confusion matrix. Unlike a typical binary confusion matrix, this confusion matrix represented the actual intent and predicted intent for all 24 intents. The diagonal of the confusion matrix can be used to see all the correct classifications where the actual intent was also the predicted intent. All the other cells of the confusion matrix represent misclassifications. For the most part, it was noted that the majority of the misclassifications, or incorrect intents, were unrecognized. There were 93 of these. As stated before, these are not the worst-case scenario as it is better for the bot to not recognize an utterance than to recognize it incorrectly. There were only two instances where two intents were misclassified more than 10 times. Intent u and m were confused 60 times and intent k and f were confused 13 times. However, since all of these intents had hundreds of test utterances run through them, none of their performance metrics were significantly affected.

**Conclusion**

In the end, these methods seemed quite promising and easy to implement with a chat bot. The main idea is to train word embedding model and then use the mean of the vectors of the words in each utterance where the vectors of the words come from the word embedding model. This is then used for the clustering algorithm. The issue that would need to be tested further is if this would work on a dataset that is more complex than that of Bitext’s dataset. As stated in the beginning, Bitext’s dataset was synthesized where it added non-real utterances for variation and repetition. My thought is that if this process was used on a raw dataset of chat utterances it would have a more difficult time creating distinct clusters.

**References**

Bitext. (2019, December 26). *Bitext’s Customer Support Dataset for free*.

https://blog.bitext.com/free-customer-support-dataset

Brownlee, Jason. (2017. October 6). *How to Develop Word Embeddings in Python with Gensim*.

Machine Learning Mastery. https://machinelearningmastery.com/develop-word-

embeddings-python-gensim/

GeeksforGeeks. (2019, May 21). *Python Measure similarity between two sentences using cosine*

*similarity*. https://www.geeksforgeeks.org/python-measure-similarity-between-two-

sentences-using-cosine-similarity/

Li, Zhi. (2019, May 30). *A Beginner’s Guide to Word Embedding with Gensim Word2Vec*

*Model*. Towards Data Science. https://towardsdatascience.com/a-beginners-guide-to-

word-embedding-with-gensim-word2vec-model-5970fa56cc92

SciKit-Learn. (2020). *sklearn.cluster.AgglomerativeClustering*. https://scikit-

learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html