July 23, 2018 Kunal Kotian

Q: How would you evaluate your autocomplete server? If you made another version, how would you compare the two to decide which is better?

The auto-complete server's performance may be evaluated by measuring the following performance metrics:

1. Percentage of messages (sent by an agent over a specified period of time e.g. 40 hours) that contained auto-complete suggestions that were accepted by the agent, averaged over several agents. A higher value would be desired since it would indicate high engagement with the auto-complete system.
2. Total number of keystrokes from agents who were given access to the auto-compete system. This would also be computed over a specified period of time, and averaged over several agents. A low value would be desirable.
3. An A/B test to evaluate the performance of a single auto-complete server:
    - Two randomized groups of agents, a control group and a test group, must be selected. The test group will have access to the auto-complete product while the control group will not.
    - An appropriate null hypothesis must be formulated, e.g. 'The difference in the mean conversation durations for the control group and the test group is zero.'.
    - After collecting a sufficient number of samples (based on power analysis), the test statistic (difference of mean durations) can be calculated and based on its p-value, we can determine if the auto-complete server provides a statistically significant improvement to the customer service agents' operation by reducing the mean duration of conversations.

    An A/B test to compare two auto-complete versions of the server:
    - For comparing two versions of the auto-complete server, we replace the control group with a randomized group of agents exposed to the second version of the auto-complete server and perform the A/B test as described in the previous point.

Q: One way to improve the autocomplete server is to give topic-specific suggestions. How would you design an auto-categorization server? It should take a list of messages and return a TopicId. (Assume that every conversation in the training set has a TopicId).

Assumption: Each message in the list of messages received by the auto-categorization server in real time is a single string.

Generating category predictions during a live conversation

- The auto-categorization server could be designed to wait till a certain number of messages or words are exchanged between the agent and the customer.
- These messages can then be used as input provided to a trained topic categorization model that returns a TopicId prediction. This would serve as the initial category prediction.
- As additional messages are exchanged between the agent and the customer, the list of input messages can be cumulatively grown and the trained model can be pinged at regular intervals to request updated category predictions. The expectation is that as more messages are provided to the categorization model, its prediction should tend to improve.
- Weight for category-specific word suggestions:
    - Using the TopicId returned by the auto-categorization server, the auto-complete server could generate suggestions influenced by the knowledge of the predicted topic.
    - The weight that the auto-complete server places on the predicted TopicId can be made dynamic – it could start off at a low weight (because initial category predictions might rely on fewer messages as input) and progressively increase the weight (and influence) of the predicted category on the auto-complete suggestions.

A reasonable approach for building the auto-categorization model would be to start with a simple model to establish a baseline and then iteratively make changes/improvements in one or more of the text preprocessing, feature extraction, and modeling steps.

Text Preprocessing

- Training data:
    - Tokenization must be performed, generating a list of word tokens for each conversation. This step generates a fitted tokenizer that can later be used to tokenize text at prediction time.
    - Some level of normalization can be incorporated to group together minor variations in language which represent similar meanings. Some examples:
        - Lower-casing all text.
        - Expanding contractions: “don’t” is the same as “do not”. Many word tokenizers in NLP libraries like NLTK and SpaCy incorporate this.
        - Lemmatizing or stemming words: Lemmatization converts a word to its base form, using knowledge of the part of speech of the word. If the processing time required for lemmatization is too high, stemming can be performed instead. Stemming converts words to their base form without using their part of speech. This results in less accurate, but faster processing. The difference in the accuracy of conversion to a word’s base form between stemming and lemmatization may not matter for this application, so it is worth testing both approaches.
    - It may help to retain special characters if they occur in product names or as a part of certain topics of discussion. Hence, it might be reasonable to avoid removing them and losing potentially valuable signals.
    - Phrase detection can be performed based on pointwise mutual information. Certain phrases might be closely associated with specific topics of conversation.
- Test data (text on which to make predictions in real time)
    - The tokenizer fitted on the training data must be used to convert each message string in the list of messages into a list of word tokens.
    - The normalization steps performed during training must be applied at test time also. If faster processing is required, sentences appearing frequently in conversations can be mapped to their tokenized and normalized forms and stored in a dict. This dict can then be accessed at prediction time to quickly retrieve the tokenized and normalized forms of common sentences.
    - If phrase detection was performed during training, the trained phrase detector model must be applied at prediction time to generated tokens that include phrases in the text. Whether the additional time required to perform this step is worth the improvement (if any) in the application’s categorization performance must be explored.

Text vectorization

- A feature vector must be computed for every conversation in the training dataset. Two simple methods can be employed for text vectorization:
  (a) Count vectorization: For each tokenized document (conversation) in the training dataset, we can construct a vector representing the number of times each token in our vocabulary appears in the document.
  (b) TF-IDF vectorization: Term frequency-inverse document frequency scores can be computed for every word token in every document. TF-IDF score boosts the weight of frequent words in a document while simultaneously penalizing words that appear across several documents. Thus, by assigning greater importance to words that are more ‘representative’ of a given document, the document’s feature vectors consisting of TF-IDF scores for each word token can improve a classification model’s performance.

- At prediction time, the vectorizer fitted on the training data must be used to transform the messages into numerical feature vectors.

Classification model

- I would prefer to build simple models to establish a baseline first. Two models I would try out (separately) for message classification are:
  (a) Multinomial Naïve Bayes, and
  (b) Logistic regression
- The trained models must be saved for access later at prediction time. Category predictions for a list of messages can be obtained from a single saved model at prediction time. Multiple different models (e.g. one naïve Bayes model and one logistic regression model) can also be trained and then stacked together using a logistic regression meta-classifier. In such a case, the trade-off between increased model complexity (stacking multiple models vs. using a single model) and the improvement in classification performance must be evaluated.

Evaluation of model performance

- False positives might lead to improper auto-complete predictions, making a significant negative impact on the agent's conversation. Hence, precision would be a good metric for evaluating the models.

Other techniques to explore

- To improve classification performance further, some of the following techniques can be employed (at the cost of increasing model complexity):
  - Classification using deep learning models like CNNs and RNNs.
  - Representing each token with word embeddings (e.g. GloVe embeddings fine-tuned on the conversations corpus).

Q: How would you evaluate if your auto-categorization server is good?

- The categorization performance of the model powering the auto-categorization server can be evaluated based on classification precision as described in the previous response.
- Additionally, for the entire auto-categorization system itself, the performance can be evaluated by simulating a conversation and obtaining updated category predictions from the server at pre-specified intervals (please see "**Generating category predictions during a live conversation**" in the previous response). The number of messages (sentences) the server requires as input to converge to correct category predictions can be used to evaluate its performance.

Q: Processing hundreds of millions of conversations for your autocomplete and auto-categorize models could take a very long time. How could you distribute the processing across multiple machines?

Using the PySpark (Apache Spark in Python flavor) framework, most of the offline processing steps – including basic tokenization, text cleaning can be performed by distributing the workload across multiple nodes. This would involve rewriting the code (sometimes simply translating to the PySpark framework, sometimes rewriting the logic for certain functions) to take advantage of Spark's distributed map and reduce capabilities. Note that the Spark ML library provides a Python API similar to SciKit-Learn for performing distributed machine learning tasks.