Classifying Customer Service Requests

Data Science-Final Project

27/10/13

Reed Johnson

**Summary:**

With approximately 93% precision, we are able to predict whether or not an inbound request will be a “Level 1” (L1) request. The classifier uses text features from the email subject and from the email body in addition to customer meta-data to make this prediction. The classifier predicts L1 on 59% of all emails that are, in fact, L1 emails, representing about a 53% decrease in time spent triaging requests for our Customer Service Reps. It will be used to partially automate a triaging process that consumes multiple man-hours daily in addition to minimizing context switching that slows all other tasks.

**Background:**

The Customer Service / Sales team at Atlassian relies on a scoring system to measure individual performance from it’s team members. Generally, each member is expected to score a minimum number of points per week. The current scoring process is manual, where every hour or so throughout the day, a team member must sort through all new requests and assign them a score. About 380 requests arrive daily during the workday, and it takes around 20 seconds to score each request. In total, this amounts to 2 hours of work time per day. The breakdown of requests is about 83% Level 1, 16% Level 2, and 1% Level 3. The goal of this classifier is to automate away as much of the triaging process as possible, so that representatives can focus on Customers rather than performance management.

**Data:**

The most obvious data available for building a classifier for emails is text data, subject and email body. However, there is also a plethora of information available in the Atlassian data warehouse describing customer-product relationships. This data includes products owned, products currently being evaluated, and previous sales to a customer. When browsing the data used for the classier, you’ll note that licenses are distinguished between behind the firewall (BTF) licenses and hosted (SASS) licenses. This is due to the fact that there are very different licensing and pricing plans between these two offerings.

**Data import:**

This was a fairly intensive proportion of the process of building this classifier. All customer data was retrieved from a Postgres database, which was fairly trivial with the exception of some basic data formatting. Extracting text data, however, was more difficult. It also lives in a Postgres database, with all email body text stored as html. Parsing out relevant text from html proved to be a challenge, accomplished with a batching job that pulled and parsed groups of emails incrementally.

**Preprocessing:**

This was the largest area of experimentation in the construction of the classifier. The basic structure of processing text data consisted of three steps: stemming words and removing stop words, creating word vectors from the features, and selecting relevant features.

Stemming words is the process of reducing words to their stems in order to better match features. There are different libraries available to do this, I tried the PorterStemmer, LancasterStemmer, and RegexpStemmer options from sklearn. For example, a given stemmer might turn both “purchased” and “purchasing” into the same token “purchas”. Stop words are common words that typically convey no meaning, such as “to” or “and”. These are removed in this step via a dictionary of English stop words. I additionally used this opportunity to make custom re-formats, such as stemming all invoice numbers, of the form “AT-######”, to “AT-“ allowing the vectorizer to recognize these patterns and turn them into features.

Creating word vectors is the process of taking the raw text and creating term document matrices. Again, there are pre-bundled functions for this in TDIFVectorizer and HashingVectorizer. TDIF refers to a term document inverse frequency matrix, which minimizes the weights of frequently occurring words in favor of less popular words that may contain more important information. The hashing vectorizer on the other hand hashes words to create raw counts. Each function also allows for normalization and feature selection packaged in. I opted for l1 normalization.

The final step of data preprocessing consisted of feature selection from the available text features. I experimented with two options. First, using term document frequency. This method ranks words by their appearance in the documents and only takes the top n terms. The other method was chi2 feature selection, which computes a chi-squared statistic for each feature and calculates a p-value used to indicate a given feature’s importance.

**Classification:**

This is a problem of supervised classification. For this reason, support vector machines and logistic regression were explored as the best options. For the most part, this consisted of training different variations of preprocessing techniques on these two models types. For each, I also explored a limited set of parameters.

The first problem that arose with the data set when fitting a classifier was the inequity in labels. When training on the full set of training data, classifiers guessed L1 for almost all requests. To cope with this, the training set of data was culled until there was an equal set of L1 and non-L1 requests in the fitting data set. This allowed the classifiers to have a better chance of finding L2 features without being overwhelmed by sheer volume of L1 features.

For the svm, I used both linear and rbf kernels. Grid search using the svm turned out to be computationally infeasible, so different feature sets and parameters were optimized individually rather than running a comprehensive search over all combinations. The linear kernel was the highest performing, so a scan over C and gamma values was performed once we’d found the highest performing preprocessing techniques in combination with the linear svm. The highest performing model used a linear kernel, C=1, gamma = 0, and used all available features. The text features were unfiltered email body and subject text, passed through a Regexp Stemmer and Hashing vectorizer. Customer features were normalized via l1 normalization. This model scored L1 requests with 93% precision and 58% recall.

Linear Regression had the bonus of training and predicting much quicker than the svm. However, an exhaustive search of parameters and feature selection wasn’t conducted due to time constraints. Despite that fact, the highest performing model was a logistic Regression model operating on text from the email body exclusively, using a TFID vectorizer, regular expression stemmer, and using only 10,000 features. This model scored L1 requests with 93% precision and 60% recall.

**Classifier performance:**

All models were trained on the months from 2013-04 to 2013-08 and test data was held out for the month of 2013-09.

The performance of a given classifier is judged on a combination of precision and recall, which can be translated in to the following criteria:

1. How often will a request have to be re-scored?
2. How many requests will we be able to auto-triage?

The models operate on curve that trades precision for recall. It will ultimately be up to managers to decide on the accuracy needed to implement any model, as there is a mental cost associated with confidence in a given value for scores. If someone is overly concerned with the score of a request, it will offset any productivity gains in auto-triaging. Alternatively, in order to see real gains, a decent proportion of emails must be auto-triaged, so recall is important as well.

To visualize performance, I’ve plotted accuracy vs. Total number of cases auto-triaged. This allows you to make an informed decision on what gains you can achieve implementing a certain classifier and with what precision.

In the test data set, about 84% of the requests were classified L1. The peak at 84% accuracy shows a naïve classifier that blindly classifies every request as L1. We see diminishing returns as we move up in accuracy. You’ll note that there is a particular stretch between 90% precision and about 93% precision where there is small decrease in emails triaged with gain in precision.

**Business Case**

As noted previously, there are about 380 requests that arrive daily and need to be triaged. As noted previously, this would take approximately two hours of work time. I’ll build a basic business case around the 93% precision point with recall of 59%. Here are the numbers:

* 315 L1 requests, 65 L2 requests - [380\*83%, 380\*17 %]
* 189 Auto-triaged requests to L1 Correctly – [315\*60%]
* 14 request Auto-triaged to L1 Incorrectly - [189/93% - 189]
* 53% of emails Auto-triaged – [(189+14)/380]
* 1 hour worth of savings

Extrapolating from this data, implementing this classifier would lead to savings of approximately $10,000 per year, or 1/8th of a full-time employee. Furthermore, the advantages to implementing this model scale with case load and should provide increasing returns over time.

**Other steps:**

There are many possibilities to further improve this model. First, more time should be spent on feature selection. The model referenced in the business case is the logistic regression model with all only email body text included. Better features certainly exist in customer databases outside of just product ownership (support requests for example) and better features are likely to arise from viewing the text in different ways.

Second, to improve the performance of the model, a more exhaustive search of parameters for the svm should be taken. This involves setting aside more time for model training or obtaining more computing power. A thorough inspection of requests classified incorrectly might give insight into potential areas to explore further.

Third, ensembling classifiers could yield great benefits. There are some models that are ultra-accurate, but only guess L1/L2 on a small subset of requests. We’ve opted to look at a more balanced model, but allowing theses models to share predictions would give the benefits of both model types.

Last, a proper implementation of this model would incorporate more recent data and use an improved cross-validation scheme for training models and searching for parameters. It is possible that testing all of the models on the same test set allowed some bias to creep into our model that would lead to over fitting.

**Conclusions:**

Implementing this classifier as is could save an hour of time spent triaging daily for Atlassian Customer Service Representatives with minimal repercussions to request scoring accuracy. Further investment in this approach will lead to even larger gains classifier performance. Even as is, the current implementation will scale with request load and overall time savings will continue to increase with time.