Using IBM Watson Cloud Services to Build Natural Language Processing Solutions to Leverage Chat Tools

Sarah Packowski
IBM Canada
Ottawa, Ontario, Canada
spackows@ca.ibm.com

Arun Lakhana
IBM Canada
Markham, Ontario, Canada
alakhana@ca.ibm.com

ABSTRACT

Chat tools are changing the way companies engage with customers. On the one hand, these tools have tremendous benefits. They can provide an excellent experience for customers who have questions or who are having trouble. Also, analyzing historical chat conversations can help a company understand customer needs and make better business decisions. On the other hand, keeping up with a large volume of live chat messages can be difficult. And easy-to-use tools for handling those messages - using natural language processing (NLP) techniques, for example, without having to build components by hand - have not been generally available. This paper describes our experience using IBM Watson cloud services to build cognitive solutions for processing chat messages. In this paper, we share five lessons we learned while building our solutions.

CCS CONCEPTS

• **Information systems → Retrieval tasks and goals**; *Decision support systems*; *Data mining*; Web applications;

KEYWORDS

IBM Watson, Natural language processing (NLP), Natural language understanding (NLU), Natural language classification (NLC), Translation, Sentiment analysis, Social media, Chat, Cognitive, Content strategy

ACM Reference format:
Sarah Packowski and Arun Lakhana. 2017. Using IBM Watson Cloud Services to Build Natural Language Processing Solutions to Leverage Chat Tools. In *Proceedings of 27th Annual International Conference on Computer Science and Software Engineering, Markham, Ontario Canada, November 2017 (CASCON'17),* 8 pages.
https://doi.org/

1 INTRODUCTION

Background

We are members of the IBM Watson Data Platform Content Design (CD) team. We care about content, about words. That includes words we write on our product user interface (e.g. embedded assistance and labels on buttons), on product web pages (e.g. marketing pages), and in formal product documentation. That also includes words about our products that we did not write, such as forum posts by our community of users, blogs by reviewers, and conversations on social media. We're story tellers and we're listeners.

Context

For the past two years, our team has been applying our CD practices to chat tools in two ways:

Sharing information with users by using chat to provide exactly the information a user needs, just when the user needs it, right where the user already is. To do this, we need to understand the user's needs and then quickly search for information.

Listening to users by analyzing historical chat conversations, looking for common problems, trends, and patterns. It is easy to use APIs to extract chat conversations, but processing the results, sometimes thousands of conversations, can be overwhelming.

Tools

Libraries that provide NLP building blocks have been available for several years. (e.g Stanford CoreNLP[6], Natural Language Toolkit (NLTK)[5], Apache OpenNLP[2], and many others.) However, for our projects we needed to move quickly, adapt to changing requirements, and often tear down our solutions after accomplishing an objective. So we couldn't afford to build components by hand. Instead, we chose to use IBM Watson natural language processing cloud services.

Our chat-processing solutions mentioned in this paper used one or more of the following Watson cloud services:

Natural Language Classifier (NLC) for classifying chat messages into predetermined groups
- Live chat applications: route messages to different teams based on message contents
- Historical chat analysis applications: distinguish social messages (e.g. "Hi!") from messages containing technical questions and problem descriptions

You can learn more about this service here: (http://ibm.biz/CASCON-2017-Watson-NLC)

Natural Language Understanding (NLU) for extracting keywords and entities
- Live chat applications: search for related documentation
- Historical chat analysis applications: cluster chat conversations by topic

You can learn more about this service here: (http://ibm.biz/CASCON-2017-Watson-NLU)

Language Translator for identifying what language chat messages have been written in

- Live chat applications: identify when chat messages are in languages not handled by the Support team so we can automatically reply asking the person who sent the message to submit their message in a supported language
- Historical chat analysis applications: identify which languages people use and roughly understand the topics of their questions

You can learn more about this service here: (http://ibm.biz/CASCON-2017-Language-Translator)

IBM Watson Knowledge Studio for creating a custom language model

We used Knowledge Studio to create product-specific custom language models based on a corpus created from historical chat conversations. The performance of our NLU solutions dramatically improved when we used custom language models.

You can learn more about this service here: (http://ibm.biz/CASCON-2017-Knowledge-Studio)

Tone Analyzer for identifying the mood of a chat message

- Live chat applications: prioritize messages that seem upset, frustrated, or angry
- Historical chat analysis applications: track sentiment over time for a product or feature

You can learn more about this service here: (http://ibm.biz/CASCON-2017-Tone-Analyzer)

What we mean by chat

In this paper, we use the term *chat* to mean any of the different types of chat tools that are available:

Live chats where users chat in real time with members of our Support team

Messaging apps where users leave messages and we reply in the chat tool or in email after some delay

Cognitive-enhanced bots help Support team members by tagging or routing chat conversations, or responding to easier chat questions with suggested links

Fully interactive chat bots trained using question-answer pairs

2 LESSONS LEARNED

In this paper, we share five lessons that we have learned about building cognitive chat-processing solutions:

(1) How do people chat?
(2) How much training data is needed?
(3) Is a trained cognitive component irreplaceable?
(4) Can you reuse a cognitive component?
(5) Are cognitive components useful for every problem?

We illustrate these lessons using natural language classifier (NLC) examples, but these lessons can be applied to most NLP, cognitive, or machine-learning solutions. Although this paper focuses on chat tools, we use the same strategies for processing other kind of messages, such as messages on social media, too.

LESSON 1: HOW DO PEOPLE CHAT?

We learned that capturing the complete first idea from a chat conversation is more complicated than just capturing the first message entered. We learned not to make assumptions about how people use chat, and instead to use historical data to guide solution design choices.

First complete idea

Chat conversation messages can be analyzed in complex ways. For example, Kim et al. (2010) demonstrated automatically classifying dialogue acts such as: CONVENTIONAL_OPENING, STATEMENT, QUESTION, and THANKS.[4]

When beginning our first chat-processing solutions, we planned a much simpler approach: use APIs to capture the first user message from chat conversations and then filter, classify, analyze, or respond based on that first message. However, when we examined our historical chat conversations, we saw two important patterns that caused us to change our plans.

Pattern 1: One idea, multiple messages

In our historical chat conversations, people rarely included a complete idea in their first message. Instead, they expressed their idea over multiple messages. Figure 1 shows this pattern.

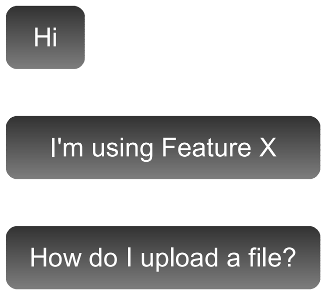


Figure 1: Using three messages to ask a question

We realized we needed to collect users' first complete idea, not just the first message they typed. But the challenge was how to know when someone is finished asking their question. For example, should we collect the first three messages? Or should we wait for five minutes from the beginning of the conversation? Or something else?

To answer this question, we analyzed 2700 of our historical chat conversations, measuring the time between consecutive user messages from the beginning of the conversation until the first reply from a member of our Support team. Figure 2 shows the results.

For most of our historical conversations, there was a delay of 30 seconds or less between consecutive user messages until the first Support team member reply. Visual inspection of the conversations where there had been a delay of more than two minutes between consecutive user messages showed that the user had often begun discussion of a new idea in the later messages.

Based on these results, we designed our solutions to consider someone to have finished typing their first idea when two minutes has passed without the user typing any new messages.

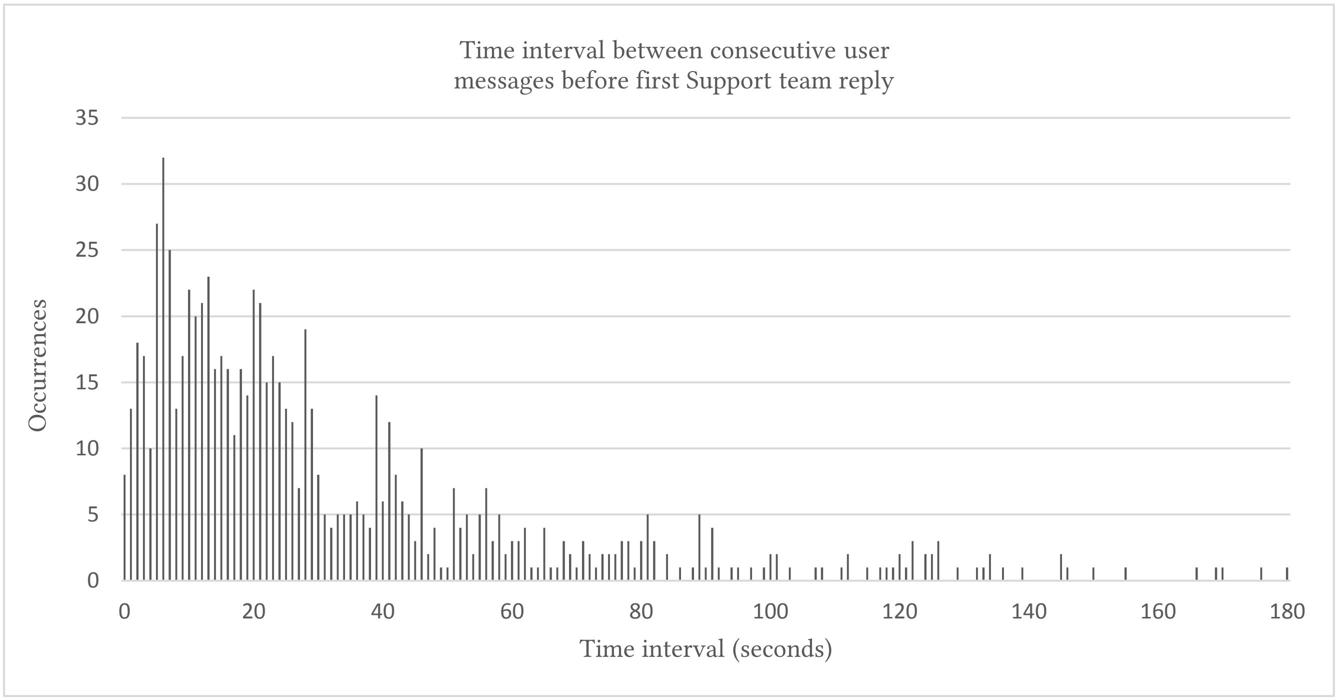


Figure 2: Time between consecutive user messages before the first Support team member reply

By listing these results here, we are not trying to imply that these numbers (e.g. 30 seconds, two minutes) are universal. Our intention is to emphasize the need to use historical chat data to drive your solution design choices, and to provide an example of how to go about doing that.

Pattern 2: Asking to ask a question

The second pattern we saw in our historical chat conversations was that some people first asked if they could ask a question, and then asked their question only after a Support team member replied to them. Figure 3 shows this pattern.

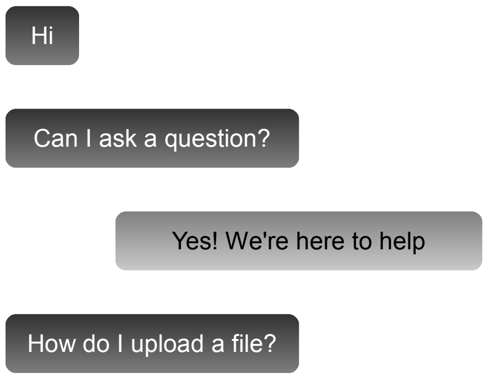


Figure 3: Some people asked if they could ask a question before asking their question

As a result, we used an NLC to identify when someone is asking about asking a question:

- Live chat applications: automatically return a message encouraging users to type their question
- Historical chat analysis applications: add logic to collect the first idea after the first Support team member message

LESSON 2: HOW MUCH TRAINING DATA IS NEEDED?

It seems obvious that more training data should be better than less training data when creating an NLC. But we learned that's true only up to a point. We also learned that factors such as the quality of the training data and the nature of the classes influence how much training data is needed to train a well-performing NLC. Finally, we learned that using an evaluation set to identify when an NLC's performance improvement levels off can indicate when more effort to collect data and continue training might not be worthwhile.

Example project: routing chat conversations

Routing chat conversations to the appropriate Support team can be done based on user characteristics, such purchase history for returning users, or clickstream data for new users. For example, Goes et al. (2011) evaluated different user characteristics-based classification systems.[3]

For a recent live chat application, we wanted to route chat conversations to different Support teams based not on user characteristics but by classifying the user messages in the chat conversation. Table 1 shows our four classes.

Table 1: Example chat messages of our four classes

Class	Example messages
Class 1 ("Hi")	"Hi!"
	"Anyone there?"
Class 2 ("Account")	"I want to change the credit card associated with my account"
	"I have a question about billing"
Class 3 ("Problem")	"Is there an issue with my server? I can't log in today"
	"My app keeps crashing. Can you help me?"
Class 4 ("Question")	"Can you tell me where there are samples?"
	"What version of JAVA is supported?"

To train our solution, we had 11 months of historical chat conversations:

(1) We used messages from the first nine months as a training set, T, to build an NLC.
(2) We used messages from the last two months as an evaluation set, E, to test the performance of the NLC.

Experiment: Tracking NLC performance by training size

To explore how much training data is needed to create a successful NLC, we performed an experiment:

(1) We created 18 training sets, $T_1, \ldots, T_{18}$, each containing a subset of the first nine months of historical data, T. The training sets ranged in size: the smallest training set, T_1, contained 86 sample messages; and the largest training set, T_{18}, contained 3901 sample messages (all the messages in T.)
(2) We used those 18 training sets to train 18 different NLCs.
(3) We tested the 18 NLCs using our evaluation set, E, to see how well the NLCs performed relative to how many sample message were used to train them.

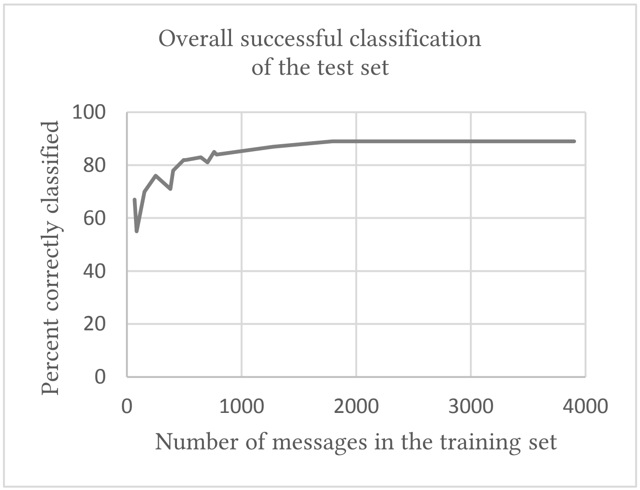


Figure 4: Overall successful classification by training set size

Figure 4 shows our results. The performance of our NLCs improved as the training set size increased up to about 1000 sample messages. As the training set size increased to 2000 messages, performance improved more slowly. Using more than 2000 sample messages did not improve performance of the trained NLC on our evaluation set significantly.

Quantity vs. quality

There's more to training a successful classifier than a large volume of training samples. After training with only 100 samples of "Hi" messages, the success rate of the NLCs at classifying "Hi" messages was nearing optimal. But it took five times longer for the NLCs to be able to classify messages in the other classes as successfully. Figure 5 breaks down the training results by class.

Figure 5 illustrates some factors that affect successful NLC training:

Distinctiveness of classes Class 1, "Hi" messages, are very different from the other classes of messages. But the distinction between a class 2 "Account" message, such as "How much does the JAVA runtime cost?", a class 3 "Problem" message, such as "My JAVA apps won't build!", and a class 4 "Question", such as "How do you build JAVA apps?", is subtle.

Uniformity within each class Class 1, "Hi" messages, is very homogeneous. But messages within each of classes 2, 3, and 4 are not as uniform. Some questions are short and to the point, for example, while some questions are long and involved.

Outliers In every few hundred of our historical messages, there was sometimes a totally unusual message like "Make $$ in your spare time from home!". There were too few of these outliers to train the NLC to classify them as a separate class. That clearly isn't a good example of a "Problem" message or a "Question" message. But including those kinds of outliers in our training data as "Hi" messages caused our NLC to misclassify real questions or problems as "Hi!" messages. In our case, we chose to remove these outliers from our training data.

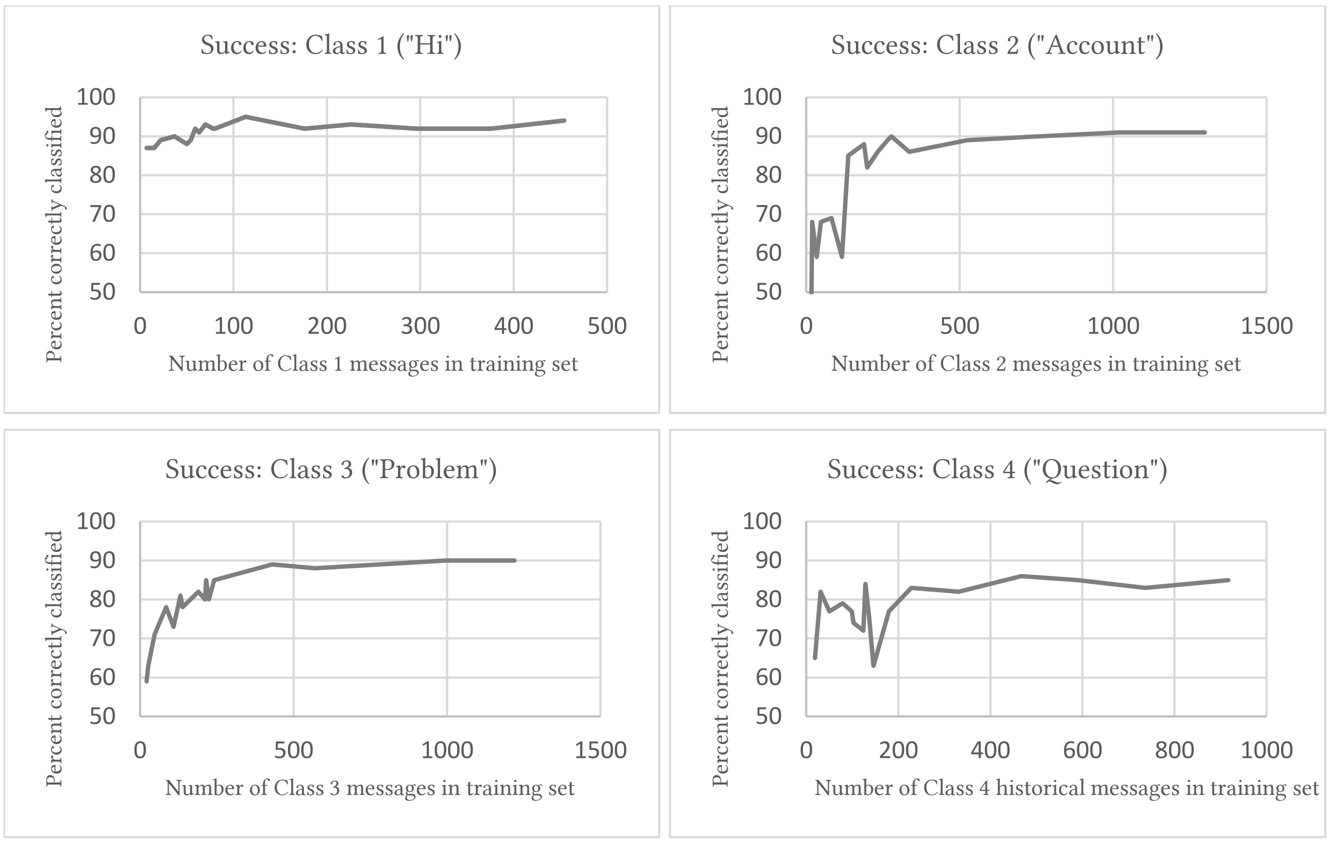


Figure 5: Successful classification of each class by training set size

LESSON 3: IS A TRAINED COGNITIVE COMPONENT IRREPLACEABLE?

We learned that we could reliably reproduce an NLC by creating a new one using the same training set.

Experiment: Comparing NLCs created with the same training set

To investigate how easy it is to "reproduce" an NLC, we performed an experiment using the same training and evaluation data described in LESSON 1, above:

(1) We created 30 NLCs, training each of them using the smallest training set, T_1.
(2) We tested all 30 NLCs using the evaluation set, E.

Figure 6 and Figure 7 show how well each NLC classified evaluation set E:

- The overall success rate varied by only 1%
- 88% of the messages in E were classified the same way by all 30 NLCs
- 97% of the messages in E were classified the same way by at least two thirds of the NLCs

These results gave us confidence that we could "reproduce" an NLC by creating a new one using the same training data.

LESSON 4: CAN YOU REUSE A COGNITIVE COMPONENT?

We learned that you cannot assume an NLC trained for one context will successfully classify data from another context.

Example: An (apparently) ideal candidate for reuse

For an early historical chat analysis project we needed to distinguish social messages, such as "Hi!" and "How's the weather?", from more technical messages in chat conversations taking place on a database manager web console. We trained an NLC to classify messages as "important" or "unimportant" using historical chat messages from that database manager web console.

A few months later, we needed to create a solution for filtering non-technical messages in real time for chats happening on an account administration web console.

It seemed like the two projects' requirements were very similar: both were processing chats from a web console, and both products were used by the same people. So we hoped to be able to reuse the NLC from the earlier project for the later project.

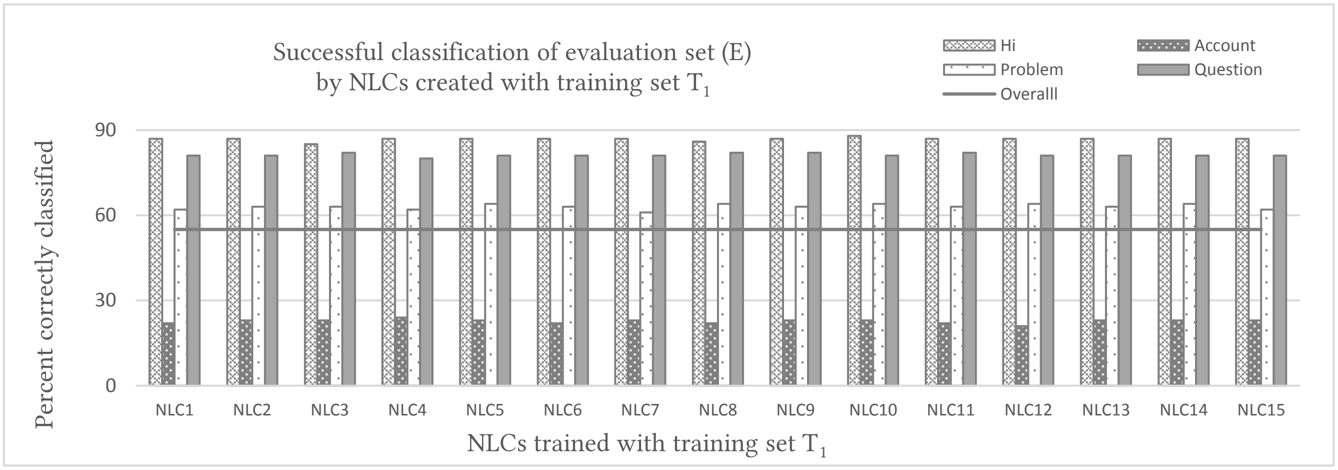


Figure 6: Performance of 15 NLCs trained with the same training set, T_1

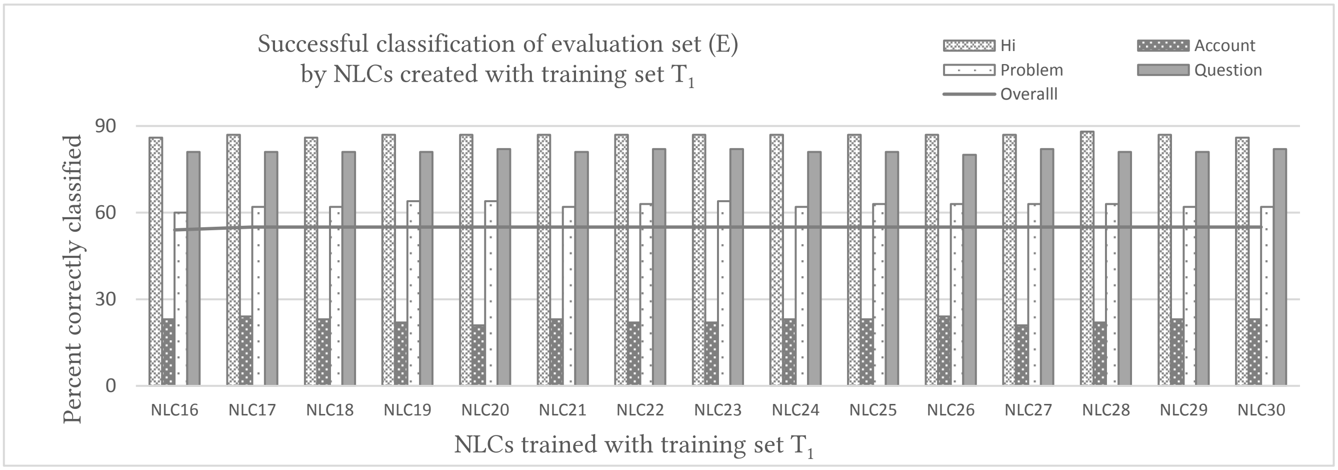


Figure 7: Performance of 15 NLCs trained with the same training set, T_1

Experiment: Testing an NLC with data from a new, different context

To explore whether we could reuse the old NLC, NLC_{old}, for the new project, we performed an experiment:

(1) We created a new training set, T_{new}, and a new evaluation set, E_{new}, using messages from historical chat conversations on the account administration console.
(2) We used T_{new} to train a new NLC, NLC_{new}.
(3) We compared the performance of NLC_{old} to the performance of NLC_{new} at classifying messages in E_{new}.

As shown in Figure 8, testing proved NLC_{old} would not work well for the new project:

False alarms both NLC_{old} and NLC_{new} incorrectly classified unimportant messages as important less than 1% of the time. This could waste our Support team members' time, but is not a critical error.

Dropping the ball NLC_{old} incorrectly classified important problems as unimportant 24 times more often than NLC_{new}. Misclassifying an important message as unimportant is a critical error, because it could cause a delay in the resolution of a user problem.

Users perform very different tasks with these two web consoles. When users of the account administration console were sending important messages, such as "I can't access my account!", NLC_{old} didn't recognize that as a technical database question like ones it had been trained with.

In software engineering, the drive to reuse components to save time is strong. However, in this example, incorrectly classifying an important message could cause a delay in the resolution of a customer's problem. We could not risk that outcome, so we did not reuse NLC_{old} for the new project.

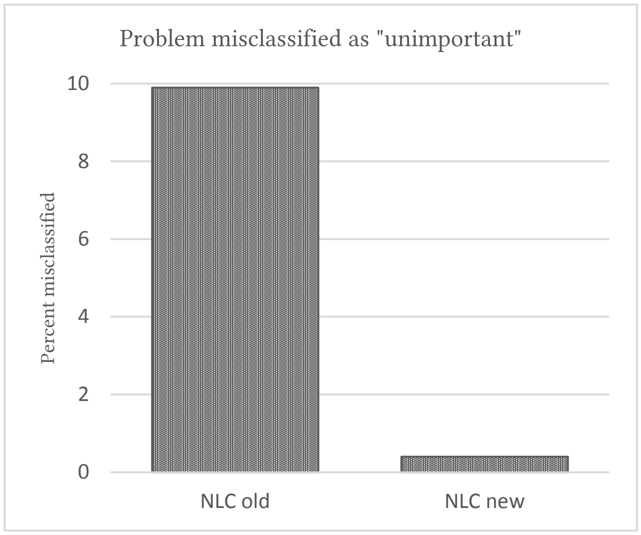


Figure 8: NLC$_{\text{old}}$ misclassified too many important messages as unimportant

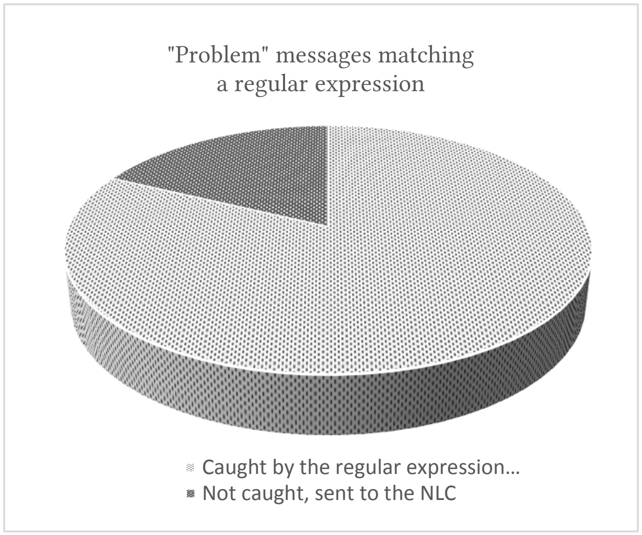


Figure 9: "Problem" messages captured by RegEx1

LESSON 5: ARE COGNITIVE COMPONENTS USEFUL FOR EVERY PROBLEM?

We learned to start simple, and then add more complex components only as needed.

Example 1: NLC vs. regular expression

For one project, we created an NLC to distinguish between "Problems" and "Questions" in chat messages. We had 2558 messages from historical chat conversations to use as training data:

- 1452 messages we manually identified as "Problems"
- 1106 messages we manually identified as "Questions"

Visual inspection of the historical chat conversations showed that certain keywords were strong indicators of problems. These keywords are captured in regular expression RegEx1:

$$/ticket|error|issue|problem|down['l]|crash/i$$

We tested our 2558 historical messages to see how many matched RegEx1. Figure 9 shows the results:

Easy catch 82% of the historical user messages that we identified as "Problems" matched RegEx1.

False alarm Less than 1% of the historical user messages that we classified as "Questions" matched RegEx1. (Example of a message that matches RegEx1 which is nevertheless just a question: "What are the best practices for error handling?")

So we adjusted our solution to use RegEx1 to catch the easy-to-identify, problem-indicating messages, and only send messages that didn't match RegEx1 on to the NLC for classification. This simple strategy reduced our API calls, speeding up our app and reducing costs associated with using the NLC service.

Example 2: Clustering vs. skimming

Project 1. About a year ago, we created a solution that displayed the most popular topics of conversation related to a particular product. Here's an overview of how we built and used the solution:

Phase 1 Collect historical data

(1) Extract historical conversations from forums, social media, and internal Support chats using APIs
(2) Save conversations in an a database for faster retrieval
(3) Identify the first idea in each conversation using heuristics

Phase 2 Train cognitive components

- Use historical conversations to train a product-specific custom language model
- Use historical first ideas to train an NLC:
  - Filter out "social" or "uninteresting" messages
  - Classify conversations as "Problems" or "Questions"

Phase 3 Production

(1) Extract conversations from forums, social media, and internal Support chats using APIs
(2) Save conversations in a database for faster retrieval
(3) Identify the first idea in each conversation using heuristics
(4) Separate different languages using Language Translator
(5) Filter out social or uninteresting messages using the NLC
(6) Using NLU, extract entities from first ideas to identify the topic of the conversation (e.g. the keywords "loading", "upload", "imported", and "ingest" are all examples of an entity we defined, called "LOAD", in our custom language model)
(7) Cluster conversations by topic
(8) Display the results for a given time interval in a graph

Figure 10 shows an example of the output of the solution. This solution was fun to build and worked well. However, it took a month to build, with multiple people contributing.

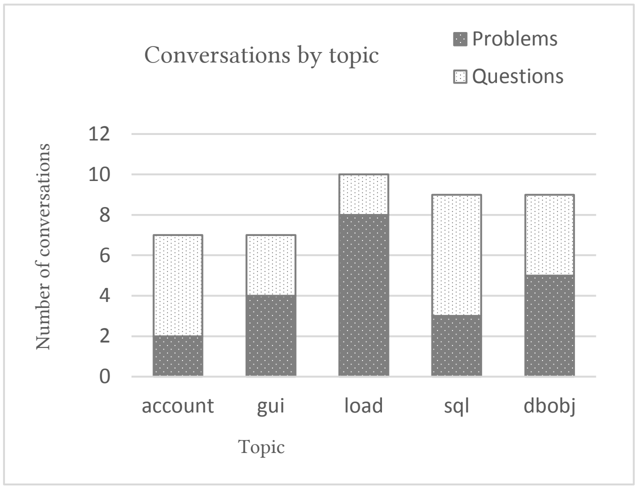


Figure 10: Clustering conversations by topic

Project 2. Recently, we needed to identify the most popular topics of conversation related to a beta program for a new product. But in this case, we had a week to build the solution, and no historical data. Here's an overview of how we approached the problem:

Phase 1 Put the following into production

1. Extract conversations using APIs
2. Save the conversations in a database for faster retrieval
3. Identify the first idea in each conversation using heuristics
4. Display the conversations in a dashboard:
    - Make the conversations easy to skim by only showing the first idea by default
    - Enable ad hoc filtering by regular expression

Phase 2 After a few weeks, we had enough data to train an NLC to filter out irrelevant messages.

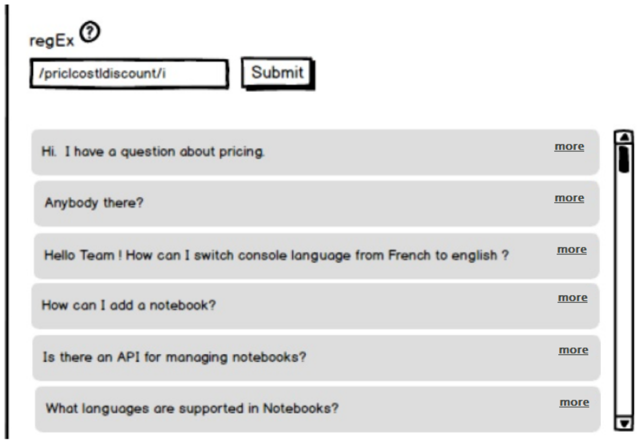
Figure 11 shows the basic design of the dashboard. This solution was quick and easy to build, and it worked well too. The product team really was able to extract valuable insight just by skimming the first ideas in the dashboard.

Comparison. Both Project 1 and Project 2 were successful and returned important insights. But, you could argue that the second, simpler project yielded greater value for time invested. Every project has different requirements. In describing these two projects here, our intention is to highlight the benefits of starting with as simple a solution as possible, and then adding more complex components as needed.

3 SUMMARY

Keeping apace with progress in natural language processing is, indeed, daunting.[1] But an emerging class of consumer software services is bringing NLP technology into the main stream. This paper shares lessons we learned while using IBM Watson cloud services to apply NLP technology to our own business challenge of leveraging chat for Content Strategy:

1. Don't make assumptions about how users chat. Use historical data to drive design decisions.



Hi. I have a question about pricing.

Anybody there?

Hello Team ! How can I switch console language from French to english ?

How can I add a notebook?

Is there an API for managing notebooks?

What languages are supported in Notebooks?

Figure 11: Simple dashboard for skimming conversations

2. More training data is better - up to a point. Tracking performance can identify the point beyond which collecting more training data and continuing training might not be worthwhile.
3. We don't need to keep every NLC we've ever created just in case we might need them. We know we can "reproduce" an NLC by creating a new one using the same training data.
4. Don't assume an NLC created for one context can be reused in another context.
5. Start simple, and then add more complex, cognitive components only as needed.

Sample code

You can download sample code from the projects described in this paper from here: (http://ibm.biz/CASCON-2017-Sample-Code). We hope the lessons described in this paper and the sample code help you to get started with these services and save you time building your own cognitive solutions.

ACKNOWLEDGMENTS

The authors would like to thank Hai-Nhu Tran and Xiangyi li, IBM Watson and Cloud Platform Content Design managers for their support and encouragement.

REFERENCES

[1] Clair Cardie. 2017. Keeping Apace with Progress in Natural Language Processing. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. 293. https://doi.org/10.1145/3018661.3022745
[2] The Apache Software Foundation. 2017. Apache OpenNLP. (2017). https://opennlp.apache.org/
[3] P. Goes, W.T. Yue N. Ilk, and J.L. Zhao. 2011. Live-chat agent assignments to heterogeneous e-customers under imperfect classification. *ACM Transactions on Management Information Systems* 2, 4 (2011). https://doi.org/10.1145/2070710.2070715
[4] S. N. Kim, L. Cavedon, and T. Baldwin. 2010. Classifying dialogue acts in one-on-one live chats. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP '10)*. 862–871.
[5] NLTK Project. 2017. Natural Language Toolkit. (2017). http://www.nltk.org/
[6] Stanford University. 2017. Stanford CoreNLP - Natural language software. (2017). https://stanfordnlp.github.io/CoreNLP/