Stack Exchange Project Report

Sam Hausmann and Jason Booth
CS 4240 - Large Scale Parallel Data Processing
4/22/2018

Intro/Motivation:

For our final project, we wanted to dig into a dataset where we could ask a solidified analytical question that would drive our process and produce clear results. With the Stack Exchange data, we wished to explore the relevance of characteristics of answers that were marked as accepted (given the elusive green checkmark by the original question poster). Additionally, since each Stack Exchange Community (there are approximately 170 with more added constantly) is its own ecosystem that caters to users interested in topics from astronomy to worldbuilding, we wanted to compare traits of accepted answers and of users across exchanges and see how they differed. Our hopes were that by performing this analysis we would be able to find some interesting results that may shed light on some differences between communities, or highlight trends for Stack Exchange as a whole. Overall this goal was accomplished, and the project gave us great exposure to Spark and other current distributed computing technologies in the process.

To select a dataset, we began by downloading portions of datasets from Reddit, Wikimedia, the US Census, and the Stack Exchange community. We decided that we liked the format and features exposed by the Stack Exchange data dump the best. We liked how the data was stored in multiple files which presented an interesting challenge, and also that there was information stored on several distinct aspects of each exchange like posts, users, badges, etc. Also, in our research and dealing with the endpoints directly we never had any problems with rate limiting or downtime, which was extremely useful for the project. A further description of this dataset is included below.

Dataset:

The first level of granularity for this dataset is separation of data by community which are in individual .7z zipped files.[1] Encapsulated within each community is a folder containing XML files for Users, Posts, Answers, PostLinks, PostHistory, Questions, Votes, Badges, and Comments. This format is consistent across exchanges except for a few cases where the file are very large. For example, Stack Overflow separated out the endpoints for each of its files, which needed to be kept track of during the extraction steps. Also, for each exchange, there is a “meta” exchange where discussions about the workings and policies of the exchange takes place. These meta exchanges have the same format, so we decided to get data from those as well, which droves up the total number of endpoints to 361. A description of the schema for each endpoint is included in Table 1.

Table 1. Raw schema skeleton for each Stack Exchange community

Badges.xml - UserId, e.g.: "420"	Comments.xml - Id

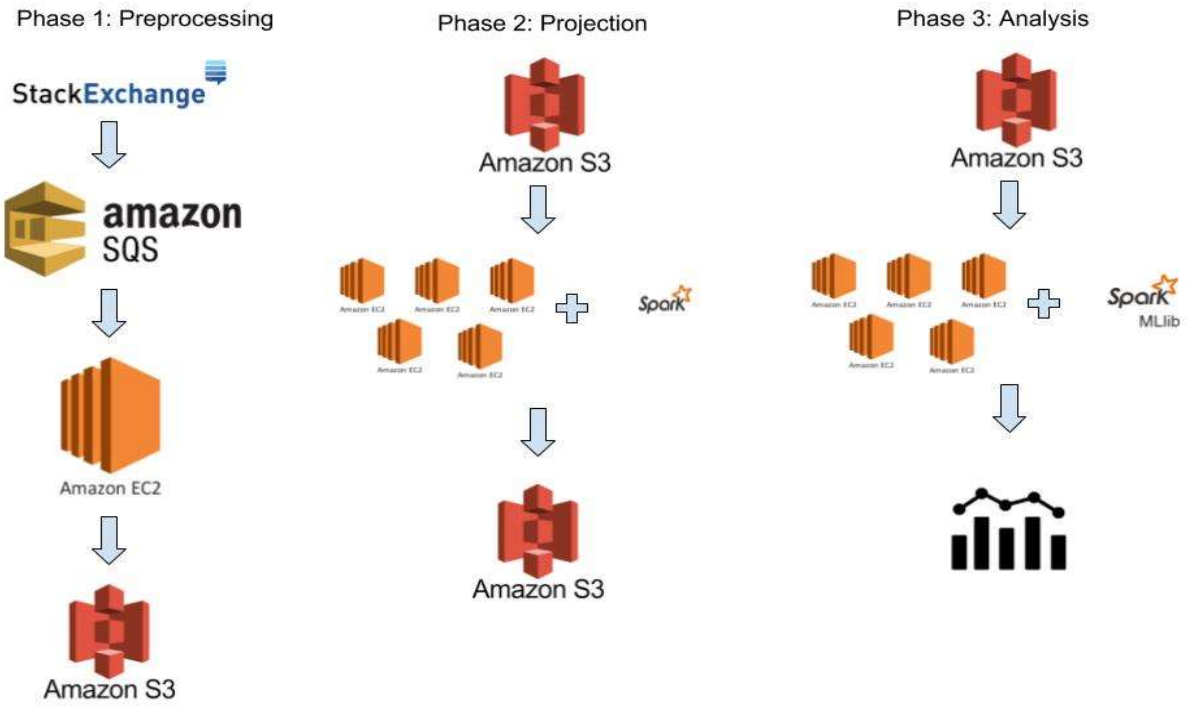
- Name, e.g.: "Teacher" - Date, e.g.: "2008-09-15T08:55:03.923"	- PostId - Score - Text, e.g.: "@Stu Thompson: Seems possible to me - why not try it?" - CreationDate, e.g.:"2008-09-06T08:07:10.730" - UserId
Posts.xml - Id - PostTypeId - 1: Question - 2: Answer - ParentID (only present if PostTypeId is 2) - AcceptedAnswerId (only present if PostTypeId is 1) - CreationDate - Score - ViewCount - Body - OwnerUserId - LastEditorUserId - LastEditorDisplayName="Jeff Atwood" - LastEditDate="2009-03-05T22:28:34.823" - LastActivityDate="2009-03-11T12:51:01.480" - CommunityOwnedDate="2009-03-11T12:51:01.480" - ClosedDate="2009-03-11T12:51:01.480" - Title= - Tags= - AnswerCount - CommentCount - FavoriteCount	Posthistory.xml - Id - PostHistoryTypeId … - PostId - RevisionGUID: At times more than one type of history record can be recorded by a single action. All of these will be grouped using the same RevisionGUID - CreationDate: "2009-03-05T22:28:34.823" - UserId - UserDisplayName: populated if a user has been removed and no longer referenced by user Id - Comment: This field will contain the comment made by the user who edited a post - Text: A raw version of the new value for a given revision - If PostHistoryTypeId = 10, 11, 12, 13, 14, or 15 this column will contain a JSON encoded string with all users who have voted for the PostHistoryTypeId - If PostHistoryTypeId = 17 this column will contain migration details of either "from <url>" or "to <url>" - CloseReasonId - 1: Exact Duplicate - This question covers exactly the same ground as earlier questions on this topic; its answers may be merged with another identical question. - 2: off-topic - 3: subjective - 4: not a real question - 7: too localized
Postlinks.xml - Id - CreationDate - PostId - RelatedPostId - PostLinkTypeId - 1: Linked - 3: Duplicate	Users.xml - Id - Reputation - CreationDate - DisplayName - EmailHash - LastAccessDate - WebsiteUrl - Location - Age - AboutMe - Views - UpVotes - DownVotes
Votes.xml - Id - PostId - VoteTypeId - ` 1`: AcceptedByOriginator - ` 2`: UpMod - ` 3`: DownMod - ` 4`: Offensive - ` 5`: Favorite - if VoteTypeId = 5 UserId will be populated - ` 6`: Close - ` 7`: Reopen	

- ` 8`: BountyStart - ` 9`: BountyClose - `10`: Deletion - `11`: Undeletion - `12`: Spam - `13`: InformModerator - CreationDate - UserId (only for VoteTypeId 5) - BountyAmount (only for VoteTypeId 9)	

Computation:

Out computation pipeline for this project can be broken down into three individual stages: preprocessing, transformation, and analysis.

Figure 1. System pipeline



The preprocessing stage consists of downloading the raw data from an endpoint, unpacking it, and storing it in Amazon S3. Because there were 361 zip files of varying size (~200K - 12GB compressed) that we needed to process, we decided to parallelize this task by offloading it to a cluster of Amazon EC2 instances rather than run the preprocessing stage on our own computers. To accomplish this we wrote two Python scripts that interacted with Amazon Simple Queue Service (SQS).

Amazon SQS allows you to initialize, populate, and empty a queue that can be pushed to and polled from by multiple machines. We decided to use this service because the preprocessing step is inherently parallelizable, and this system design is extremely scalable. With several processing nodes continuously polling the queue for messages, all someone would need to do to process more exchanges would be to add to the queue and wait for the uploads to finish.

The first python script read a text file containing all of the HTTP endpoints for the Stack Exchange data, created messages with the endpoints and respective community names in the body, and pushed those messages into an SQS queue. Our second Python script was created with the intention of running it on multiple machines at the same time. It connects to our existing SQS queue and continuously polls for messages. After a message is received, the Python script runs a series of syscalls to wget, extract, and push each Stack Exchange community sub-dataset into an Amazon S3 bucket. After each dataset is unpacked and put into S3, the message from the queue is deleted and the machine continues to poll. We ran five EC2 instances at once to achieve parallelism with this task and finish it faster than it would have taken if we performed it serially. This process took approximately a day to complete.[2]

After downloading, unpacking and hosting our dataset, we needed to project it down to a reasonable size for analysis and remove any superfluous fields that would not be considered in our analysis. We sat down with a printout of the raw schema (see Table 1. above) and went over each field to hypothesize whether or not it might be useful as a feature influencing an accepted answer or if it could be used to create a derived feature in our analysis. After deliberating we came up with a list of features below which would comprise our projected down dataset.

Table 2. Feature table

Feature	Explanation
Score	The difference between upvotes and downvotes
ViewCount	Number of views a post has
BodyLength	The length of the body of a post
CommentCount	The number of comments on a post
FavoriteCount	The number of favorites on a post
TimeSinceCreation	Amount of time passed between question genesis and when an answer was posted
SumCommentScore	The sum of all the scores of all comments on a post
LinksCount	The number of links in the body of a post
AcceptedByOriginator	Denotes the answer was given a green checkmark. This is what is used in our regression.
Offensive	A metric that indicates that a user has marked an answer as offensive
Favorite	A metric that indicates that a user has favorited an answer for easy reference
Reputation	A user's representation of how trustworthy you are in your community. Accrued through upvotes, suggested edits, bounties

UserCreationDate	The date a user account was created
Age	The age of a user
AboutMeLength	The length (in words) of a users about me
UpVotes	Number of upvotes on a post
DownVotes	Number of downvotes on a pose
Suffrage	Badge awarded for voting on many questions, answers, and comments
Electorate	Badge that indicates a user has voted on 600 questions and 25% or more of total user votes are on questions
Civic Duty	Badge that indicates a user has voted 300 or more times
Explainer	Badge that indicates a user has edited and answered a question within 12 hours of each other and the answer's score is above 0
Refiner	Badge that indicates a user has edited and answered 50 questions with both actions within 12 hours of each other and each answer's score is above 0
Nice Question	Badge that indicates you posted a question with a score of 10 or more

The data processing stage of this project is written in Spark and scala, and used spark-core, spark-sql, and spark aws libraries. You must pass two command line arguments in order to run the second phase of the project -- the name of the S3 bucket to connect to and the name of the Stack Exchange Community whose data will be utilized. At the beginning of our project we attempted to use spark-xml to parse in the xml for us, but during the timeline of our project, support for self closing xml tags was under development and was not yet released. Instead, we used scala's native xml library and xpath expressions to read in the files and parse out the desired fields individually.

In order to project our data down into a reasonable size consisting of only the aforementioned fields, we utilized case classes of each XML document (Users, Users, Posts, Answers, PostLinks, PostHistory, Questions, Votes, Badges, and Comments). For our computation we were not ingesting any attributes of the PostHistory file, so we decided not to upload that document for each community to S3. Our case classes consisted exclusively of the features we chose above and those required for derived attributes, so they did not contain all of the data shown in Table 1. We also wrote a base class for utility functions such as creating a file path to the specific document in our S3 bucket and parsing specific data types (ie dates).

One cool computation that we did regarding derived features is the creation of columns for the badges dataframe. For this step, we wanted to use an aggregateByKey instead of a groupByKey which is known to produce overhead for shuffling of data. To do this we added a

parametrized Utility object which has methods for adding to and joining list buffers. This is important because buffers allow for inexpensive creations of lists, whereas dealing with pure lists would cause overhead in creating new lists for each added object. After the counts for each type of badge we were interested in were totalled, we converted the object into a dataframe using a schema defined in the Badges class. Other calculations of derived features can be seen in the code on github.[3]

On initial local trials with very small exchanges, the code ran very slowly and we could see in the run logs that there were many shuffles occuring during execution. After looking through the slides from the course, we suspected that the shuffles were caused by joins on our dataframes, and that co-partitioning would be a great option to increase speed. We also inspected the number of partitions being used for each dataframe and were very surprised to see that even with the smallest dataset, each one had 200 partitions. After researching this, we found that if two dataframes are not co-partitioned, spark will set the new number of partitions after a shuffle to a value defined by *spark.sql.shuffle.partitions* which defaults to 200. To increase the execution speed, we set out to reduce the size of the final dataframes as much as possible before joining, and to make each dataframe co-partitioned during a join to reduce shuffling.

Our end goal for the data-processing stage was a dataframe where each row represents a single answer and each column represents an attribute of that answer that we could feed into a logistic regression model. We knew that the number of answers would be the size limiting factor for all of our other dataframes that we create. Therefore, we decided to filter down all of the other dataframes as fast as possible to only objects that were related to an answer, or objects that contain information about users who actually wrote an answer. To accomplish this, we read in and cached a post RDD object, which contains all of the questions and answers for a given exchange, and created a list of answerIDs and userIDs (for users that wrote an answer) that we needed for analysis. Then we passed these lists to each of our functions which created dataframes for other xml files and filtered all of their data to only those that were contained in the list. For example, for the badges.xml file we filtered down all of the objects to only badges for users that had posted an answer. This downward projection of the dataframes before our joins really helped to reduce the size of these objects and speed up operations like repartitioning.

As previously discussed, we knew that the number of answers would be the size-limiting factor for all of the other dataframes. Because of this, we decided to use the number of partitions for the posts RDD to set the *spark.sql.shuffle.partitions* parameter for all of our dataframes. That way when doing joins, all of our dataframes could be co-partitioned appropriately. Also, prior to each join, we repartitioned the relevant dataframes on the same column so that they would be co-partitioned to reduce shuffling. With these optimizations our code sped up immensely, on the order of a ten times speed up. After this processing the data was written to a csv file and put in our S3 bucket for phase 3 of the project.[3]

One of the largest issues we had while deploying this program dealt with a structural error that arose when distributing the program across a cluster. When we distributed our program to a cluster of Amazon EC2 instances, we uncovered an error telling us to “set master URL”,

which was odd because it was working fine locally. We learned that this is because we were setting our Spark context outside the scope of our methods that utilized the textfile and parallelize functions, and that during execution they were referenced when they had not been initialized. After troubleshooting this for a long time, we fixed this by initializing the spark context and placing all of our helper functions that relied on spark sql and spark context inside of main, so that everything would be referenced in the same scope.

In order to take advantage of Apache Spark's parallelism, we needed to run our program across a cluster of computers. We chose to use Amazon EC2 to spin up our nodes because we had several hundred dollars of credit towards AWS services from Amazon's promotional student pack. In order to avoid some of the boilerplate tasks that come with setting up a cluster intended to run a distributed Spark program, we utilized Flintrock which requires you to fill out some specifications (EC2 key-name and file path, EC2 ami, region, instance type, number of worker nodes, whether or not you want HDFS and Spark installed on each node) in a YAML file. For our cluster we ended up with four m4.large worker instances and one driver instance to project down most of our datasets. The computation time for these datasets was extremely small, several minutes at most. However, our Stack Overflow dataset which was approximately 61GB in size would fail every time we ran a job on it.

After scouring the internet for solutions as to how to maximize our usage of our current nodes, we ended up taking a three pronged approach to increase our performance and successfully run our computation on the Stack Overflow data. First, we requested a special order limit increase from Amazon so that we could use six m4.10xlarge nodes (160GB RAM and 40 cores) that would comprise our cluster. Then we suspected that caching our Posts RDD at the beginning of the program and calculation of lists of AnswerID and UserID could be causing memory issues for our system with so many users and answers. We also reasoned that the number of partitions that were being used for our dataframes was probably over 200 already, so the default value for setting the number of partitions after a shuffle was not decreasing performance like it was for the smaller datasets. For these reasons, we decided to get rid of the caching, calculation of lists, and resetting of the shuffle partitions parameter.[4]

Next, we researched online to figure out how to find the optimal number of executors, number of cores per executor, and memory per executor so that we could best take advantage of parallelism across each node. We learned that if we had too many executors for a spark job, there would be too much overhead in the coordination of the executors. If too few executors were used, we wouldn't be taking advantage of enough parallelism in the program.[5] After all of these parameter changes and using the larger sized machines, the processing of the stack overflow data completed and took approximately 45 minutes.

The third stage of our project consisted of setting up and running a logistic regression model on our data. We set this stage up so that it accepts command line arguments for the bucket it would be reading from, and a number of exchanges to run the analysis on at once. Spark's machine learning library offers a very clean API to define a pipeline for your data. For this project the pipeline consisted of an assembler to define and extract the features from our

dataframe on which to run the model. Then a scaler was applied so that all of the features would be on the same scale. This is mathematically important for our analysis because if the features are on different scales then the coefficients for each feature would also be scaled different and you would be unable to compare them directly. The pipeline ends with the logistic regression model, which outputs a trained model that we can use to predict a score for accuracy on our test dataset.

Another step in this phase is the calculation of the correlation matrix for all of the input features. This is important because an underlying assumption of a regression is that there is no strong correlation between the input features (generally above .8 is considered strong correlation). If the correlation is too strong, a phenomenon called coefficient splitting can occur where the true significance of a feature ends up being split between two features, and each one individually looks insignificant in terms of the model. Luckily, we did not find there to be a strong correlation in the trials that we ran on, so our coefficient values should be mathematically sound.

The third stage in our pipeline was much less difficult to deploy and took the least amount of time to execute compared the other stages. This is because the projected down data was on average about 20 times smaller than the data passed into phase 2. The stack exchange data was reduced to a size a little bit over 3 gigabytes and took around 7 minutes to analyze. The results of this stage were simply written to the console because the only results we needed were the features and coefficients.[6]

Ultimately, because of how long the extraction stage and the troubleshooting in phase 2 took we accrued approximately $400 in AWS charges. Part of this was due to human error (leaving the cluster running overnight), and part due to an extraction failing or a Spark job failing and having to restart several times. With the $200 dollars of free credit Amazon gives out for students, each of us will pay approximately $100 dollars in total charges, which we consider approximately the same as the price of books or materials purchased for other classes.

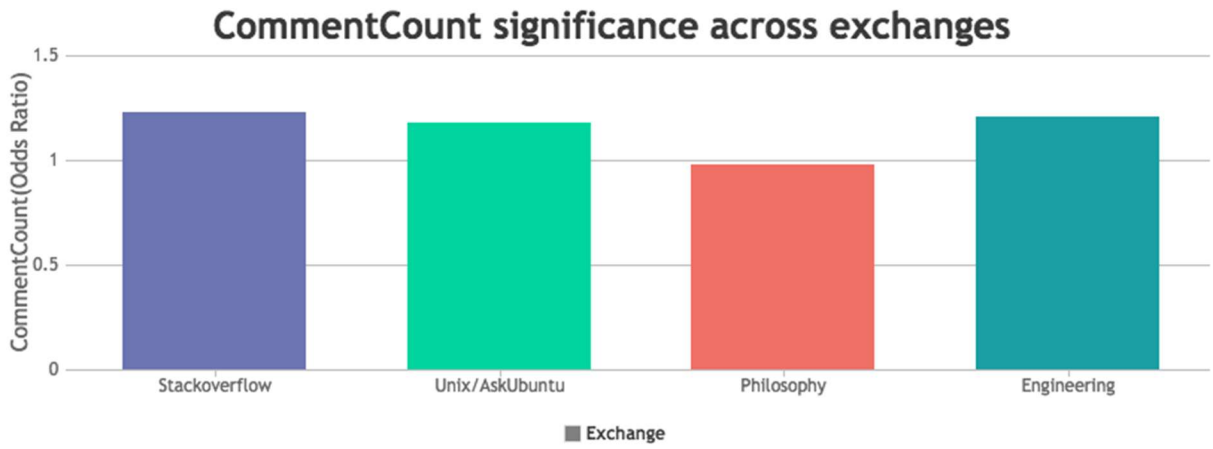
Analysis:

After deriving our features, projecting our data and running a logistic regression on it, we found that some features were more indicative of an accepted answer than others. It should be noted that in order to compare our results from the logistic regression on a linear scale, we took the odds ratio of each result which consists of performing the computation □□□□□□□□□□□□. An odds ratio is interpreted as the increase in odds per unit increase of the feature (scaled). This means that an odds ratio of 1 has no change in odds for being the selected answer. A ratio above 1 is positive, and a ratio below 1 is negative.

For our comparison of results, we chose to use coefficient data that we collected from four combinations of the largest Stack Exchange communities. We felt that Engineering and Stack Overflow were two prominent communities that are relevant to us as Computer Engineers. Unix and AskUbuntu are two large subcommunities that we decided to combine together in the analysis because of their similarity. Philosophy was a very different exchange from the others and we thought that it would add variety to the analysis.

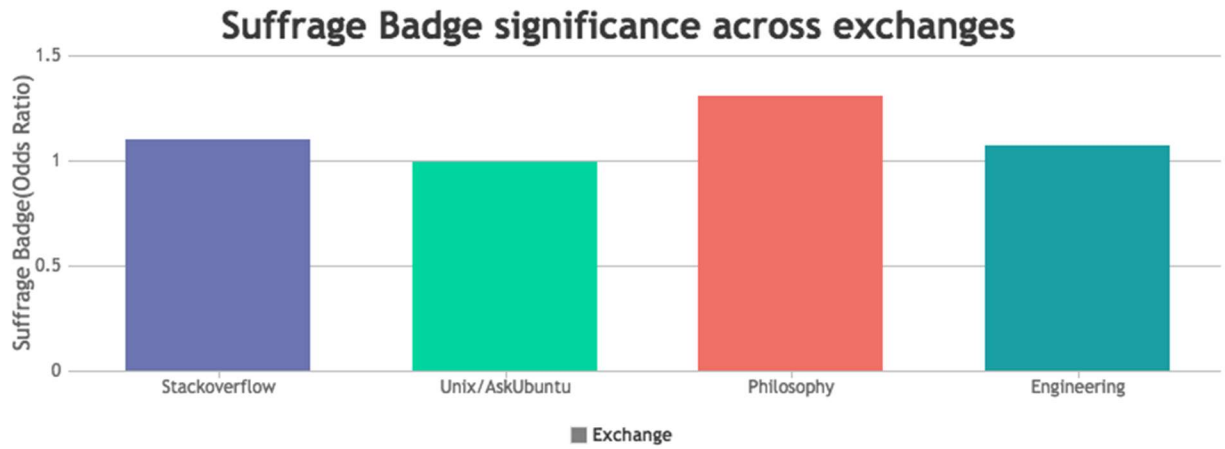
The first feature that we found to have an impact on accepted answers across all four communities was the CommentCount of an answer. We found that in the Stack Overflow, Unix/AskUbuntu and Engineering communities, the presence of a large comment count had a positive effect on an answer being marked as accepted. In the Philosophy community however, we found that the presence of a large comment count on an answer had a negative effect on an answer being marked as accepted. We hypothesized that users across the three STEM related communities encouraged controversy and discourse related to an answer while users of the Philosophy exchange were more apt to accept an answer when there is little controversy or discussion.

Figure 2. CommentCount significance



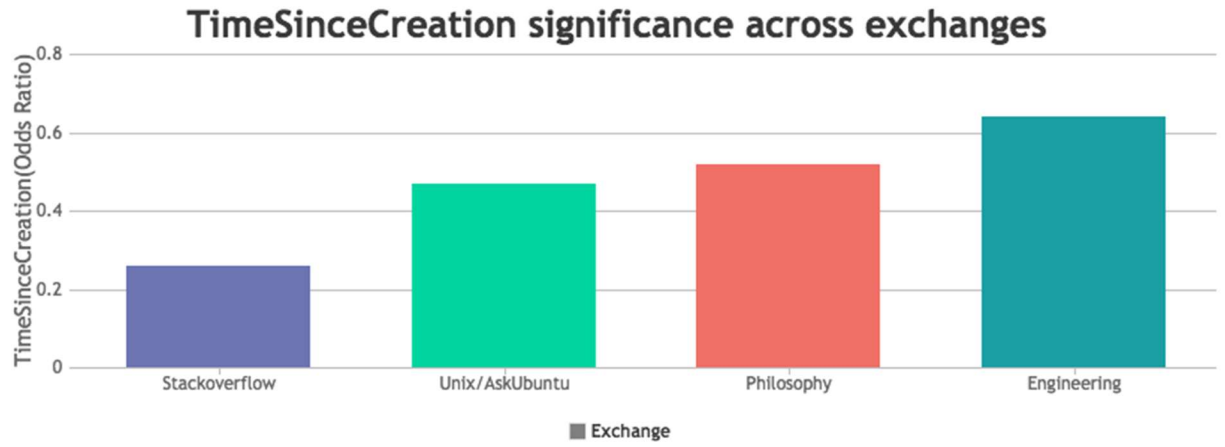
Another feature we found to have a significant impact on the Stack Overflow, Philosophy, and Engineering communities with respect to users with accepted answers was a user having earned the Suffrage badge. This badge indicates that a user consistently votes on questions, answers, and comments making them an active member of the community. This generally can be interpreted as signifying that more active members in the community for philosophy tend to get more selected answer, whereas this is not the case for Unix/AskUbuntu. For Engineering and Stack Overflow, the results are somewhere in the middle.

Figure 3. Suffrage Badge significance



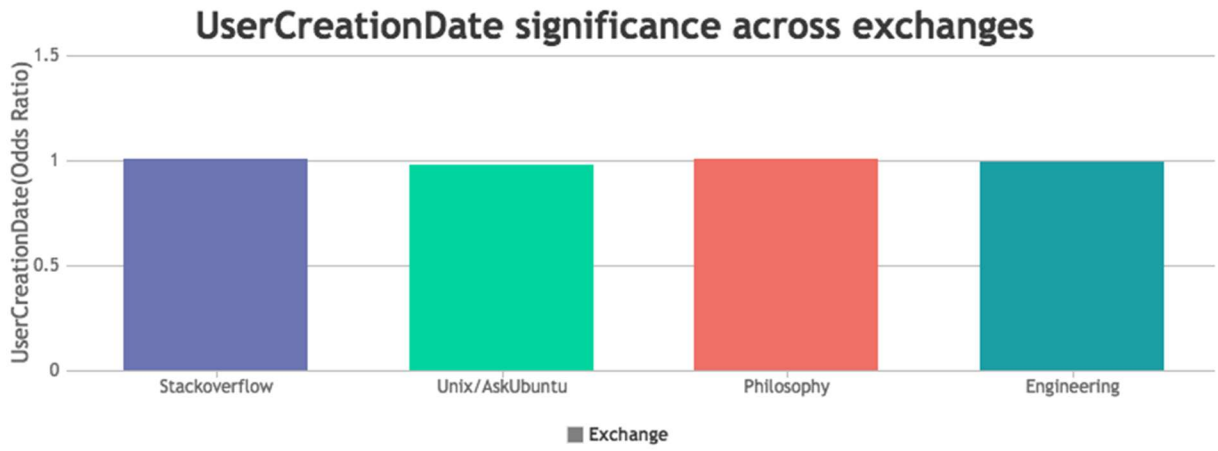
We chose TimeSinceCreation as a feature that we felt would likely have inverse significance on an accepted answer. We guessed that answers that were posted soon after the question was posted were more likely to be accepted. The graph in Figure 4 shows this to have been found true. An odds ratio of less than 1 for this feature across all four communities shows that the greater the time span between question and answer genesis, the less likely the answer will be accepted.

Figure 4. TimeSinceCreation significance



Finally, at the beginning of our project we thought that users who have existed longer and built up a reputation would have a significant advantage when it comes to answers being accepted. What we found, visualized below in Figure 5, is that for the most part, being a long standing member of the community has little effect on whether your answer will be accepted or not. This is really good news for new users because it means that anyone can theoretically post selected answers for questions and contribute to the community.

Figure 5. UserCreationDate significance



Conclusion:

Overall this project has been a great experience for both of us and gave some great exposure to Spark and the field of distributed computing. There were some places where we struggled along the way, but these really helped us to understand issues that are core to understanding how Spark operates and can affect performance in real world application. Getting to see real results at the end and attempting to interpret them in the context of our original goals was a really cool experience that provides us with interesting conclusions that we can carry forward and discuss with others that may want to hear about our project. As stack exchange users ourselves, we also intend to keep an eye out for people posting questions about Spark or SQS. If it is related to what we worked on in the project, we'll try to post solutions to share the knowledge and help other people. Hopefully they'll be selected.

Links/Citations:

1: Link to data used for the project:
https://archive.org/download/stackexchange
Notes on the organization of the github repo:
For this project we used a bit of an unusual git workflow. We decided to use a branch for each of the stages for the project. This was done to maintain the whole code base under a single repo, while discretizing the code for each of the stages. We intend to transition the repo into three separate repos after the project is over, however, to maintain commit history we decided to keep it this way for the final write up. Also, it should be noted that the DataCleaning branch is regarded as the development branch for the data cleaning step, while the s3_scala branch is what was used mostly for changes required during deployment of the DataCleaning branch. The cleaned, commented and local only version of phase 2 (citation 3) is at the tip of the s3_scala branch, while an old commit (citation 4) is linked to show what was actually deployed on Stack Exchange Data.

2: Link to the phase 1 github repo for the project (master branch):
https://github.com/SamHausmann/stack_exchange_analysis/tree/master

3: Link to phase 2 github repo used in class. Local only, commented version.
https://github.com/SamHausmann/stack_exchange_analysis/tree/s3_scala

4: Link for project that was deployed on spark for stack overflow dataset (removed caching, filtering on lists, etc).
https://github.com/SamHausmann/stack_exchange_analysis/tree/f68bbb409e45d97e9c1a77f959ded3e94c76e8a9

5: Write-up of tuning execution parameters for Spark jobs by github user "spoddutur"
https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html

6: Link to Phase 3 of the pipeline (local version, running on a single dataset on a subset of the extracted features. Used during the presentation demo)
https://github.com/SamHausmann/stack_exchange_analysis/tree/LogisticRegression