In [2]:

```
from IPython.display import HTML, display
```

RaceRaves.com Race Recommender

NLP-based recommender engine built using race reviews from the RaceRaves.com website

Problem Statement:

Runners are always looking for new challenges and setting new goals for themselves. But it's not always easy to find that next race that will help you make that goal. The information available online is scattered and inconsistent. RaceRaves.com does a great job of collating race information and collecting reviews from racers. They also have a great Find A Race feature, but it really just tells you which races are coming up that meet the criteria you select. And those criteria are currently limited to Distance, Terrain, Geography and Date:

https://raceraves.com/find-a-race/ (https://raceraves.com/find-a-race/)

Once you have the results, you can sort them by date, overall rating or alphabetical. But there may be other factors that determine which race you want to run. For example, you could be looking for a flat course or a scenic route. While RaceRaves could try to anticipate all the factors that might matter to a user, another option is mining the review data to determine what factors matter most to each racer. And once you've established that, you can look for races with reviews that talk about those same features. That is the goal of the recommender engine.

Data:

All of the data for this project was scraped from the RaceRaves website. I began by scraping all races dated from June 2017 through May 2018 using the Find a Race page and setting monthly criteria. I then compiled the user IDs from the race reviews posted to these races and pulled the individual racer data for all users collected from the race pages. The final counts were as follows:

- Unique Racers = 2,009
- Unique Races = 2,103
- Total Reviews = 6,414

All of the post-scraping code for this project can be found here. (https://git.generalassemb.ly/dwaynejarrell/DSI-Capstone/blob/master/Capstone%20Recommender%20Final.ipynb)

Data Cleaning

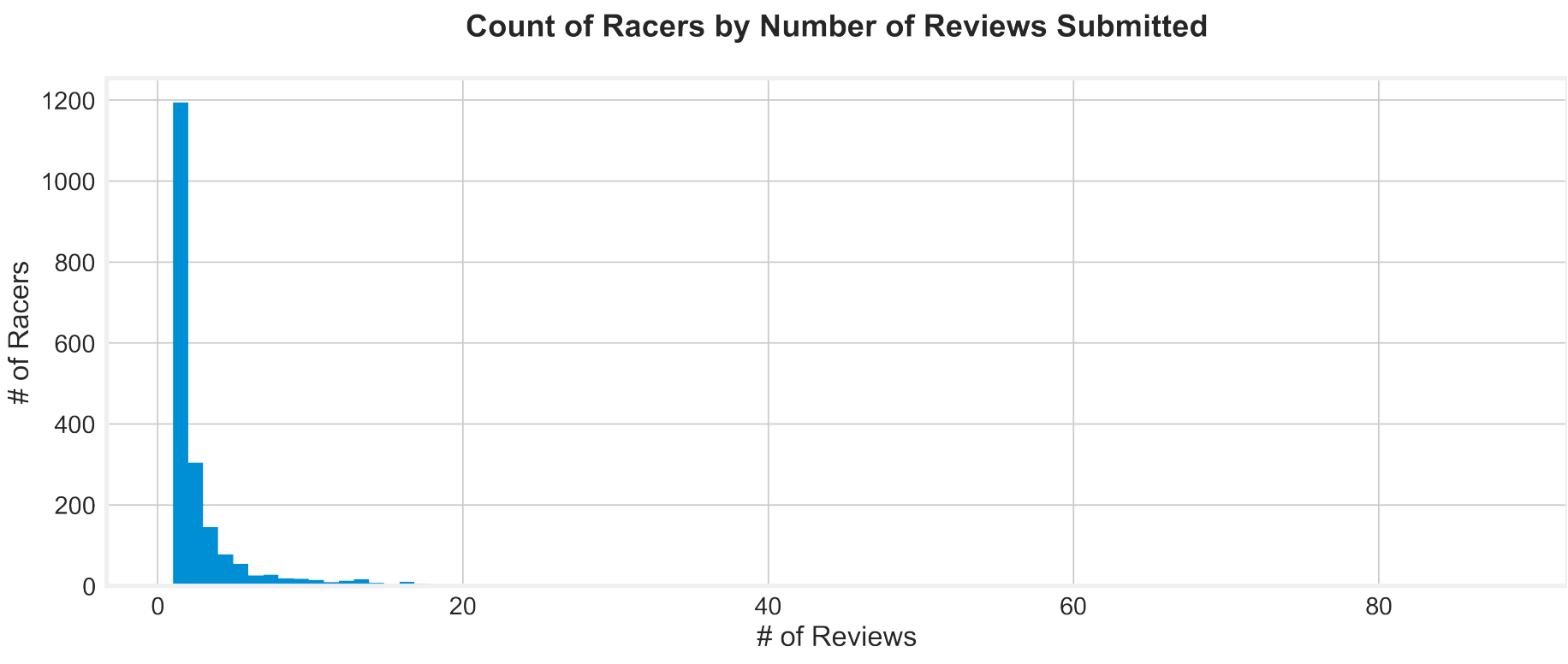
Data cleansing and formatting was built into the scraping process, so there wasn't a lot of cleaning to do once the data was in Python. One exception was the Affiliations feature, which had some old values that didn't correspond to the current options available on the website. For example, some racers had an affiliation of "Ironman athlete", but the current option on the website is just "Ironman". I created a function to update the three old values.

Because the core data came from racer pages, I started out with one row per RaceRaves user ID. Each row could have multiple race reviews, which were stored in a dictionary when scraped from the website. So the first major step was splitting out the reviews into individual rows. I did this by parsing the dictionaries and creating a unique row for each of the 6,414 racer/race combinations with reviews.

After splitting out the data for each review, I noticed that the number of race distances represented in the data was a bit unwieldy - there were a total of 135 different distances, with 58 of those having only 1 review. I decided to limit the the distances to those with at least 20 reviews. That gave me 15 distance categories. All other distances were lumped into 'Other'.

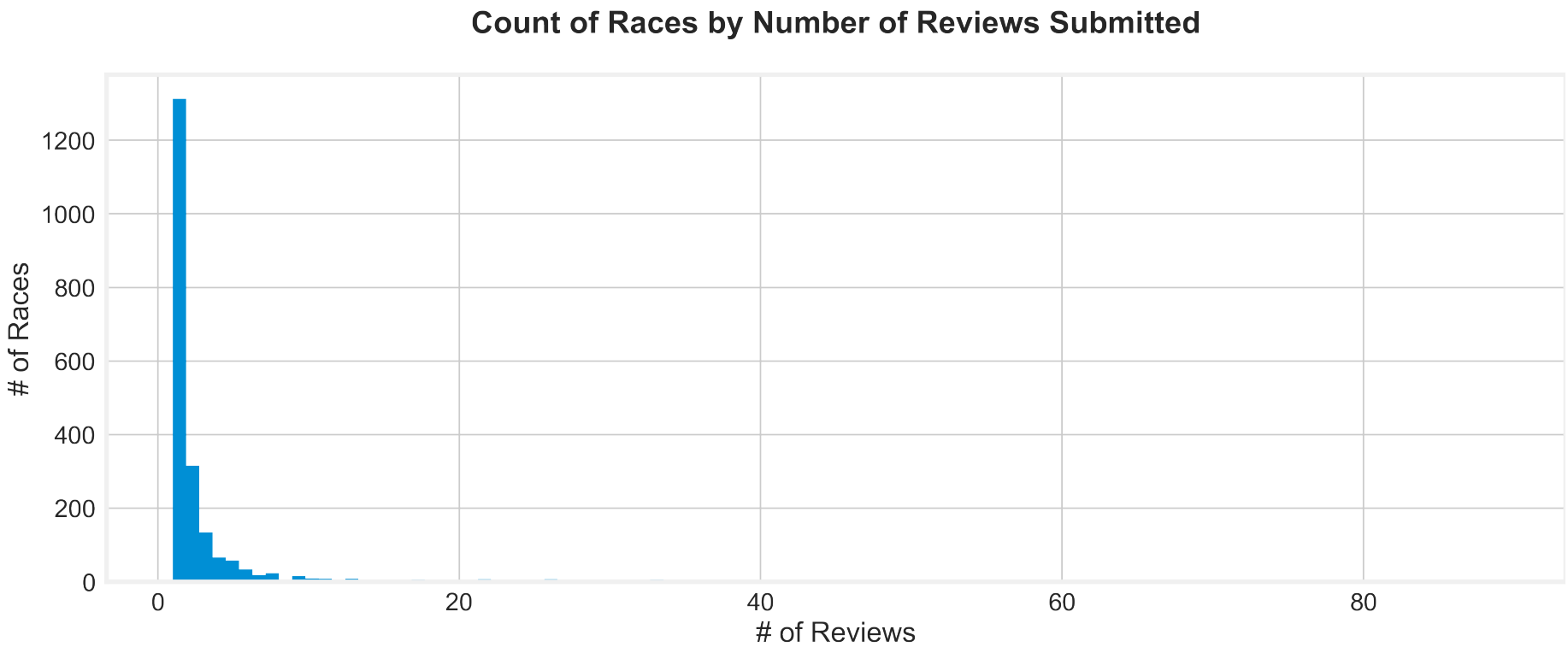
Exploratory Data Analysis

The first step was to get a sense of review frequencies for both racers and racers. Given the nature of the data collection and compilation, each racer was guaranteed to have at least one review. A quick check told me that the average number of reviews was 3.2, but the median was 1 and the max was 88. So we clearly have a skewed distribution, as you would expect:

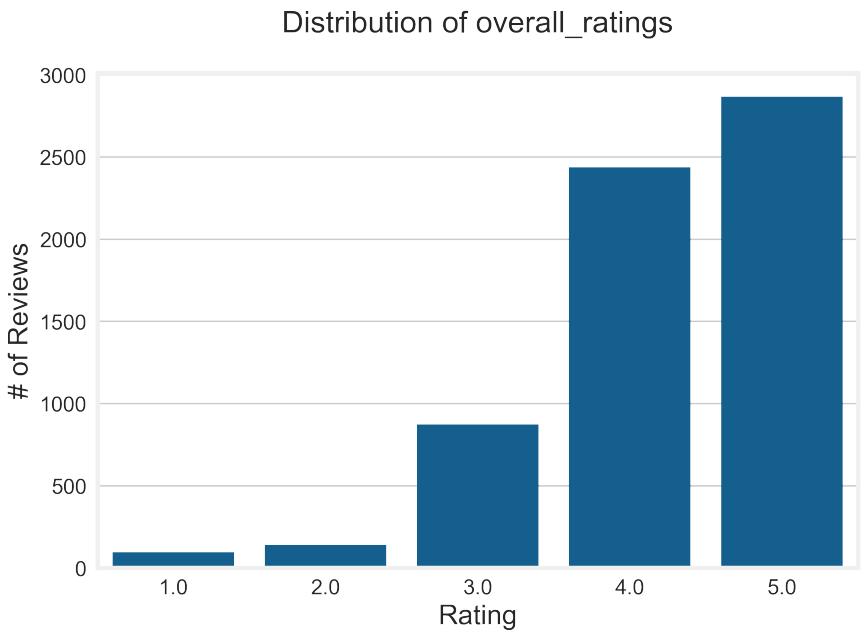


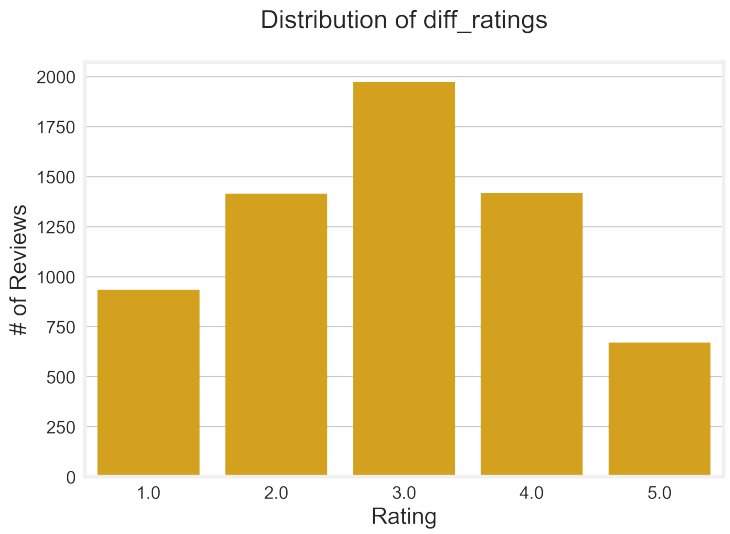
Unfortunately, 59% of the racers (1,195 of 2,009) have only 1 review, but even that one review should give us information on what matters to the racer.

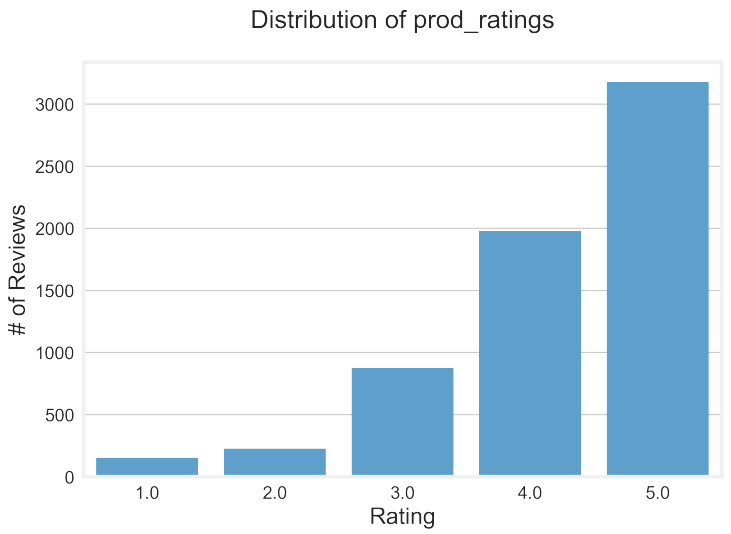
The story for races is similar - 62% (1,131 of 2,102) have only 1 review. Average number of reviews per race is 3.05, and the max is 89.

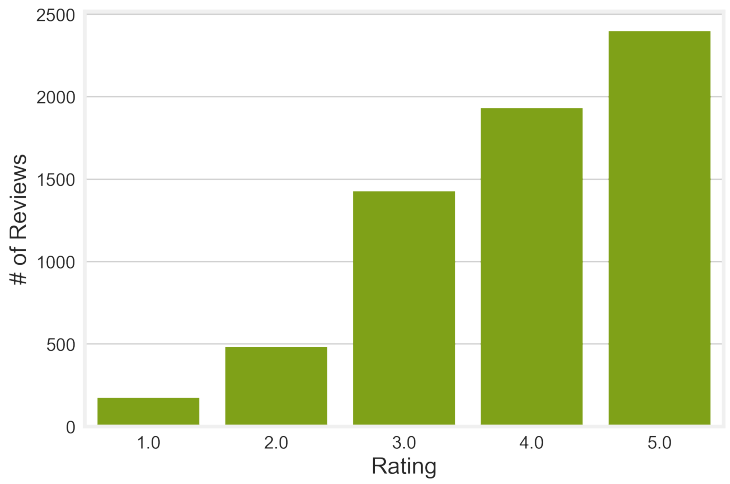


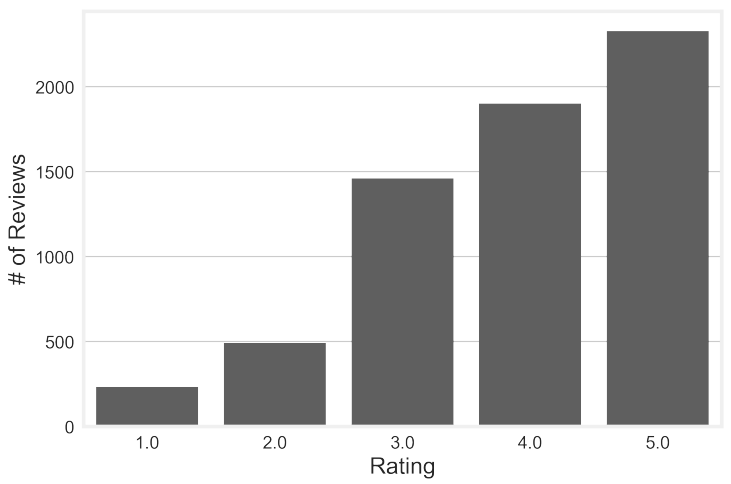
Next, we can look at the ratings. There are five total for every review: Overall, Difficulty, Production, Scenery and Swag. There are some clear differences in the distributions.





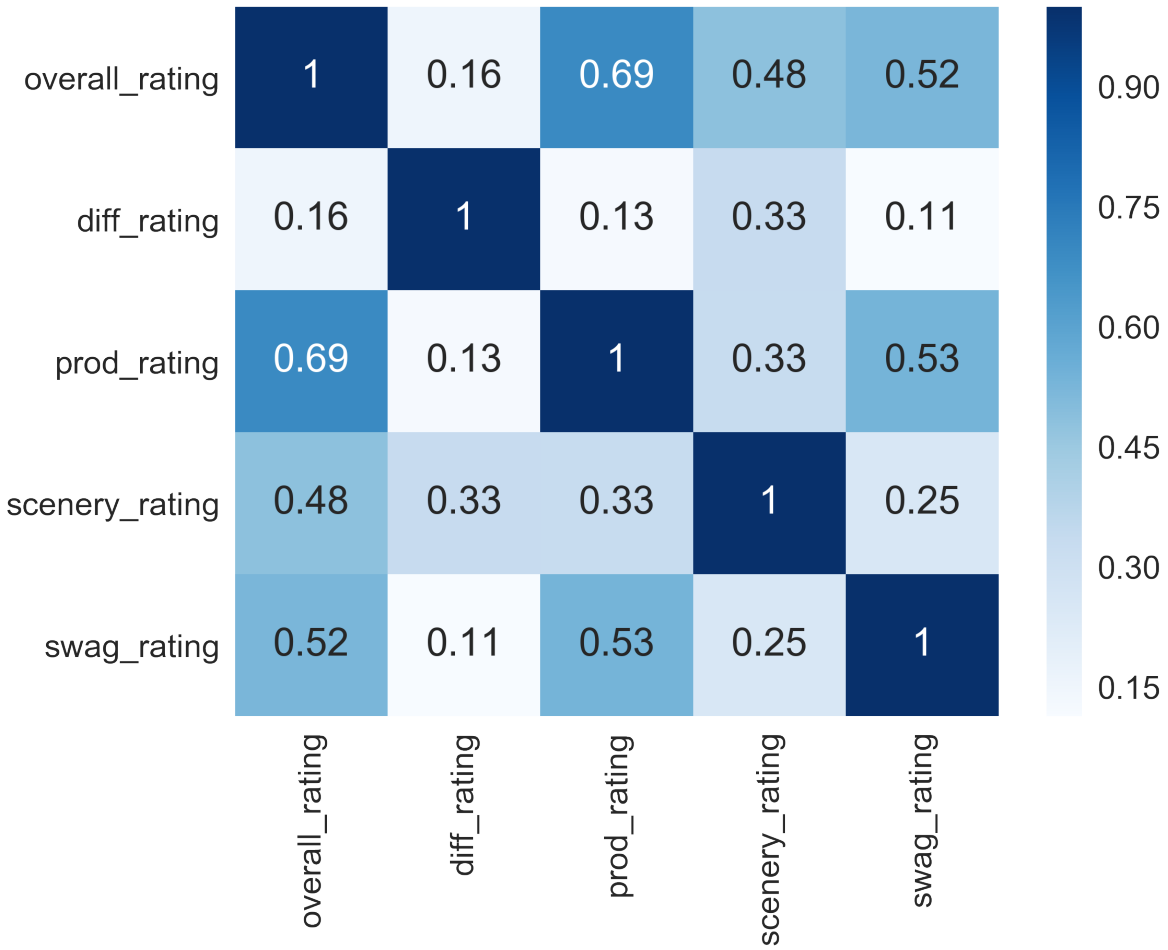






Overall Rating and Production Rating both have means of 4.2, but Production is more likely to be rated a 5 - 50% of reviews gave a production rating of 5, but only 45% of overall ratings were a 5. Scenery and swag are more likely to get rated a 3 than overall and production, but they still get 5 ratings more than any other value. Difficulty, on the other hand, is unlikely to be rated 5 - the average difficulty rating is only 2.9.

Next step was a look at the correlation matrix for all of the ratings:



Clearly, production ratings are the most highly correlated with overall ratings, but scenery and swag are also correlated at 0.5. Difficulty ratings have a very weak relationship with the overall ratings. If we run a simple linear regression to predict overall ratings with production ratings, we get an R-squared of 48%. So we can say that production ratings are driving about half of the variance in overall ratings. If we add the other three ratings to the regression, R-squared only goes up to 58%.

Feature Engineering

Because I am using NLP to build the recommender, feature engineering was focused entirely on the processing of the words in the reviews. This can be broken down into three key parts: n-gram selection, stop words, and stemming.

N-gram selection

Based partly on prior examples of recommenders built from reviews and partly on exploration of the review data for this project, I decided to focus exclusively on bi-grams for all NLP analysis. In particular, the patterns and frequencies of bi-grams corresponded more directly to the themes a runner might include in writing reviews than single words did.

Stop words

After several iterations of running the count vectorizer on the raw review data, I chose to add the following to the standard set of Engligh words provided in sci-kit learn:

- marathon
- because
- mile
- join
- ultra
- ive
- takes
- all numbers (generally used to denote distance, which is captured elsewhere in the data)

Stemming

Instead of stemming words with a standard tool, I did my own analysis of similar words that represent the same basic term or concept. For example, while the term "aid stations" was clearly the most common bi-gram across all reviews, some people used the singular "aid station" or called them "water stations" or "water stops". There were also mutltiple versions of "start" and "run". To address this, I set up a dictionary to map replacement words.

Modeling

Validation

In order to provide some validation of the races recommended to RaceRaves users, I chose to create a holdout sample of 20% of users. The reviews for these users were excluded from the training so I could apply the recommender to them and assess the hit rate for races they've already reviewed. Thus, the first step of modeling was to randomly select the 80% of users who would be used to train the recommender.

Counts for the training vs. test data were as follows:

Training

- Unique Racers = 1,607
- Unique Races = 1,768
- Total Reviews = 5,057

Holdout/Validation

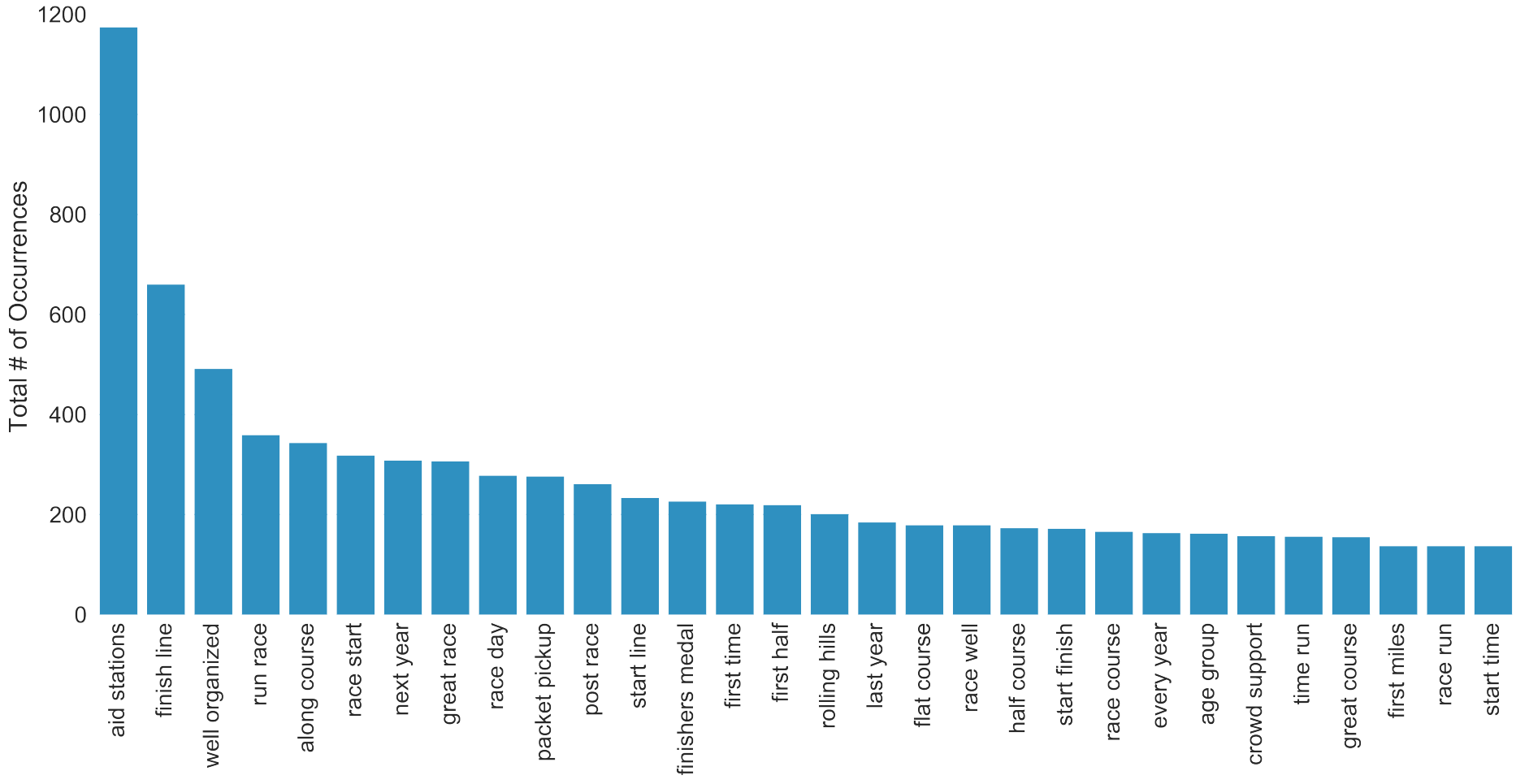
- Unique Racers = 401
- Unique Races = 776
- Total Reviews = 1,357

Count Vectorizer

The first step in the build process was the count vectorizer. Because LDA is expecting word counts, I went with the standard CountVectorizer in sci-kit learn. (I tried TF-IDF, but the results were considerably worse.) Final options selected to maximize hit rates were as follows:

- n-grams = 2 (bi-grams only)
- stop words set to custom list created above
- minimum term frequency set to 25
- maximum term percentage set to 0.25

After running the count vectorizer, which gave me 517 bigrams, I compiled the top 30 bigrams to review and validate as being relevant to racers.



Latent Dirichlet Allocation

The next step was topic assignment using the LatentDirichletAllocation module within sci-kit learn. I tried a wide variety of parameters and evaluated them all with the hit rates. The final parameters were as follows:

- Number of topics = 30
- Learning Method = Online variational Bayes method, which uses mini-batches to update the topics
- Learning Decay = 0.8 (default is 0.7)
- Learning Offset = 10 (default)
- Maximum # of Iterations = 100

In addition to checking the hit rates for training and test data, I reviewed the bi-grams across the topics to ensure that the resulting themes made sense. Here are the top 20 bigrams for each of the 30 topics chosen by the model (in no particular order):

Matching Racers and Races

Assigning Topics to Racers

The first step in matching racers and races was to assign the topics to all of the racers in the training data. In order to do that, I rolled up individual race reviews at the user ID level and created one bag of words from all the reviews for each user. For the purposes of clustering and profiling later, I also collected the number of reviews and number of distances, created dummy variables for each of the distances run and merged in the user's affiliations and average ratings.

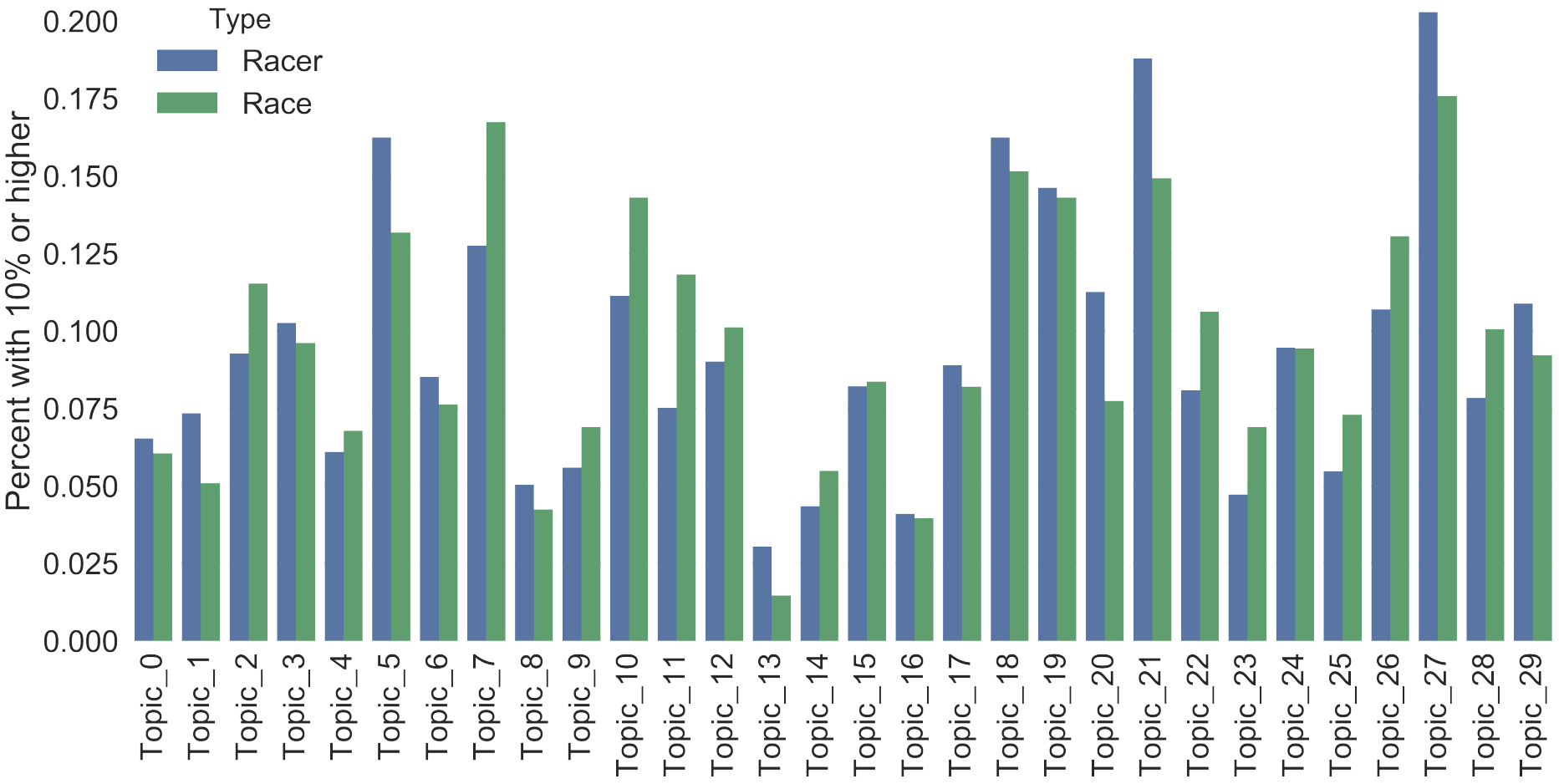
Once I had the racer-level data, I ran the fitted Count Vectorizer and LDA models on the combined reviews. Before fitting the LDA, I divided the counts by the number of reviews to account for the fact that the model was fitted on single reviews. For users with multiple reviews, this means a single bi-gram would be downgraded while a mention of that bi-gram across all reviews would give it full weight in the LDA model.

Assigning Topics to Races

Next step was assigning the topics to all of the races in the training data. This followed the same basic process as the racers. I rolled up individual race reviews at the race name level and created one bag of words from all the reviews for each race. I also collected the number of reviews and distances, created dummy variables for each of the distances available in the race and merged in average ratings.

As I did with the racer data, I ran the fitted Count Vectorizer and LDA models on the combined race reviews. I also divided the counts by the number of reviews to account for the fact that the model was fitted on single reviews, just as I did for racers.

Here's a look at how the topics fell out by Race and Racer, using a 10% threshold to determine topic assignment:



Matching Racers and Races

For each of the 1,607 racers, I calculated the differences in topics between the racer and all 1,768 races using Euclidean Distance. Euclidean distance is the straight-line distance between all 30 topic probabilities, calculated using the Pythagorean Theorem (square root of the sum of squares). I then ranked all races for each user and chose the top 5 recommended races

Hit Rates

Training Data

In order to measure the efficacy of my race recommender, I let the recommender choose races that the racer may have already run and reviewed. From a practical perspective, this makes sense - if a racer liked a given race, he or she is likely to want to run that race again. They would likely be happy to see the race show up as an option, but this is a choice that RaceRaves and/or the user could make. Including these races also allowed me to calculate a Hit Rate, which is the % of times the recommended race is one the person has already run and reviewed.

The hit rates for the training data are below:

- Number of matches for Top Race = **182** (**11.3%**)
- Number of matches Top 5 Races = **358** (**22.3%**)

Note that these hit rates are biased upward by the fact that the same reviews were used to build the recommender and score the races. We need the validation set to truly assess the success of the recommender.

Validation Data

To validate the recommender, I ran the same process on the 20% holdout racers that I did on the training racers - roll up reviews to racer level, apply the count vectorizer, assign topics using LDA, and then calculate distance and rank races. It is important to note that the set of races I matched to (the same 1,768 used above) contained NO REVIEWS from these RaceRaves users - all of their reviews were held out during the initial random selection. So any matches we find are based on matching these users' review topics to the topics from other users' reviews.

Here are the results:

- Number of matches for Top Race = **9** (**2.2%**)
- Number of matches for Top 5 Races = **29** (**7.2%**)

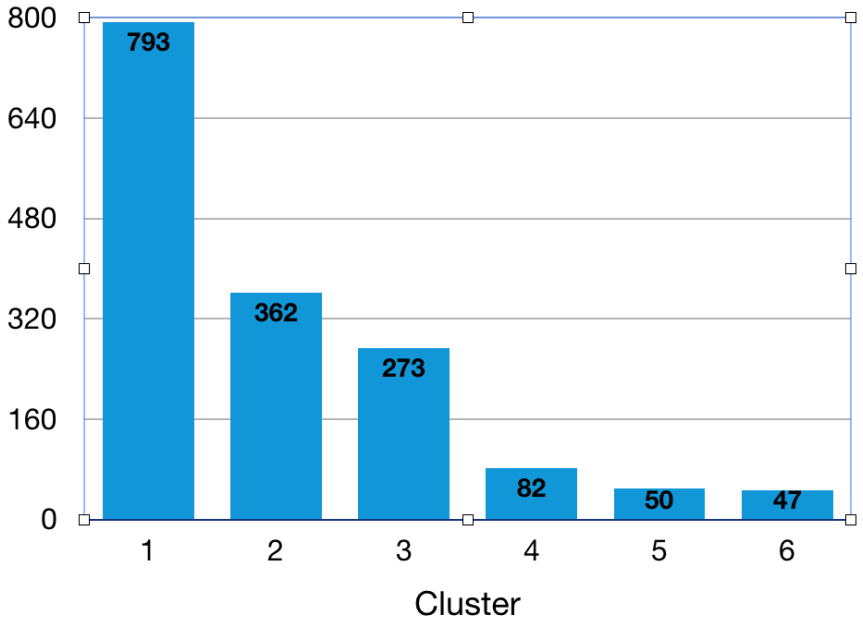
It's also true that excluding the holdout reviews meant that nearly half of the races reviewed by this population (334 out of 776, or 43%) were not included in the training data. There were only 442 races that existed in both the training and holdout data. The true hit rates would be more than double if those races could have been included in the training. All of which indicates that the recommender is doing a good job of matching up racers with appropriate races.

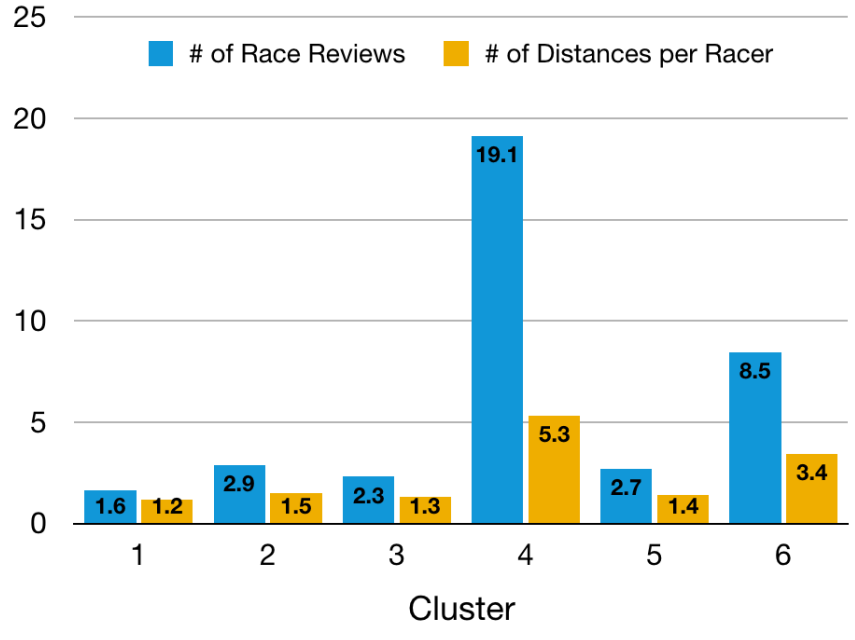
Clustering

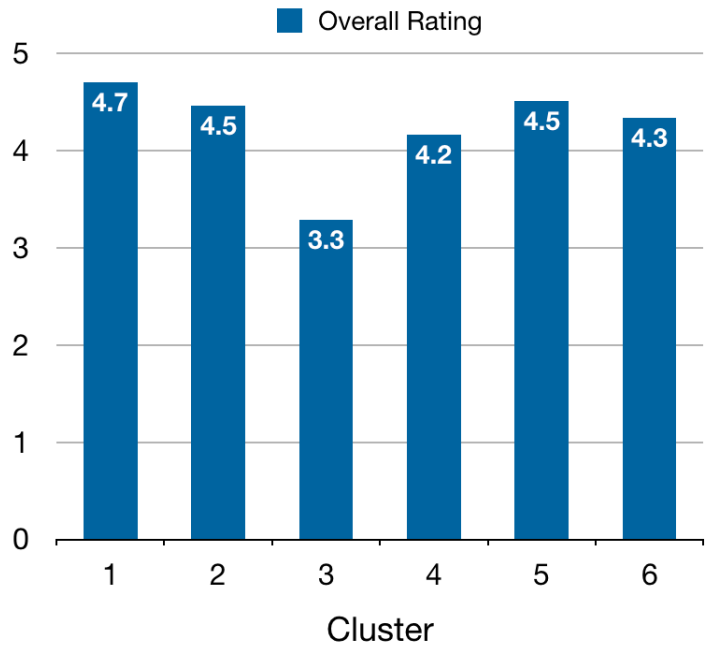
In addition to completing the initial phase of the Race Recommender, I wanted to take a look at the users of RaceRaves.com and understand their review/rating behaviors and see if we can break them out into recognizable clusters. I tried a few methodologies, but the one that gave the best results was K-Means clustering using the following profile features:

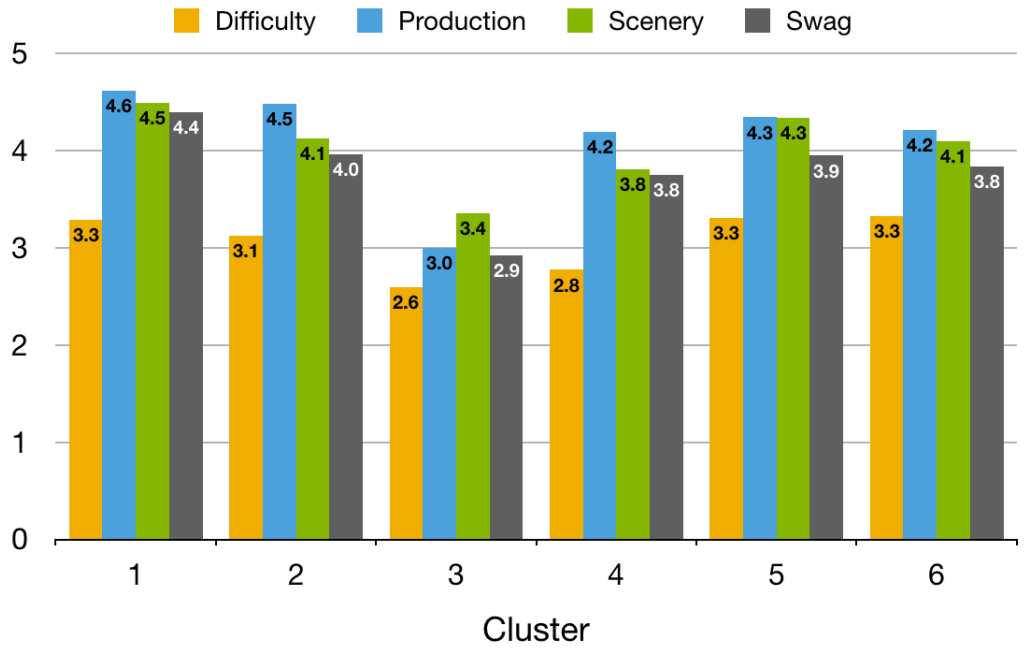
- Number of Reviews
- Number of Distances
- Average Ratings (all 5 factors)
- Distances run (Marathon / Half Marathon / 10K / 5K / 10 Miler / 12K / 15K / 50K / Other)
- Affiliations
- Topic Probabilities

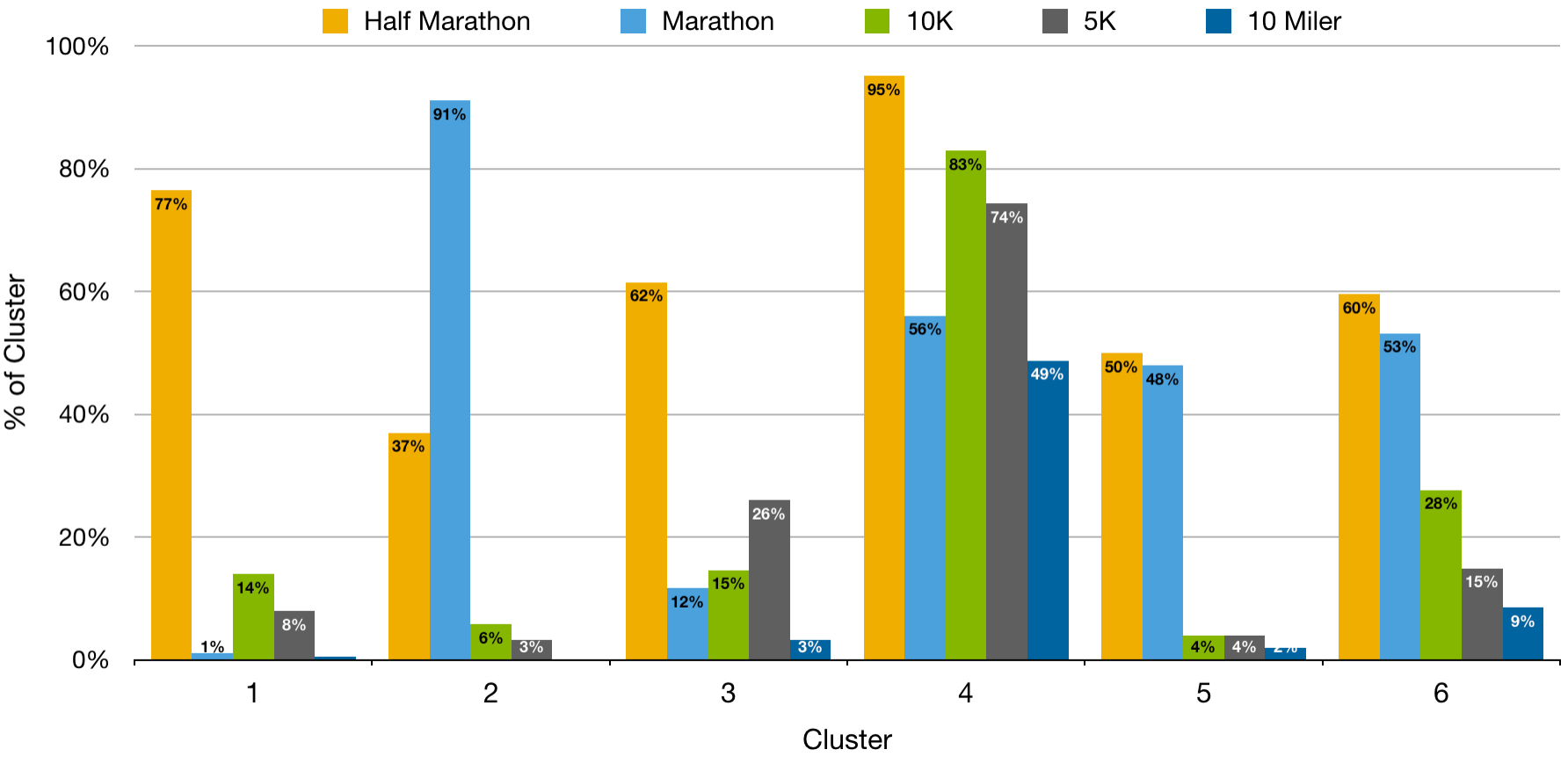
The final specifications I chose, based on the relative size and cohesion of the cluster, were 6 clusters, 25 initial centroids and max iterations of 300. I also standardized all of the features prior to fitting the K-Means. The following charts summarize the resulting clusters.

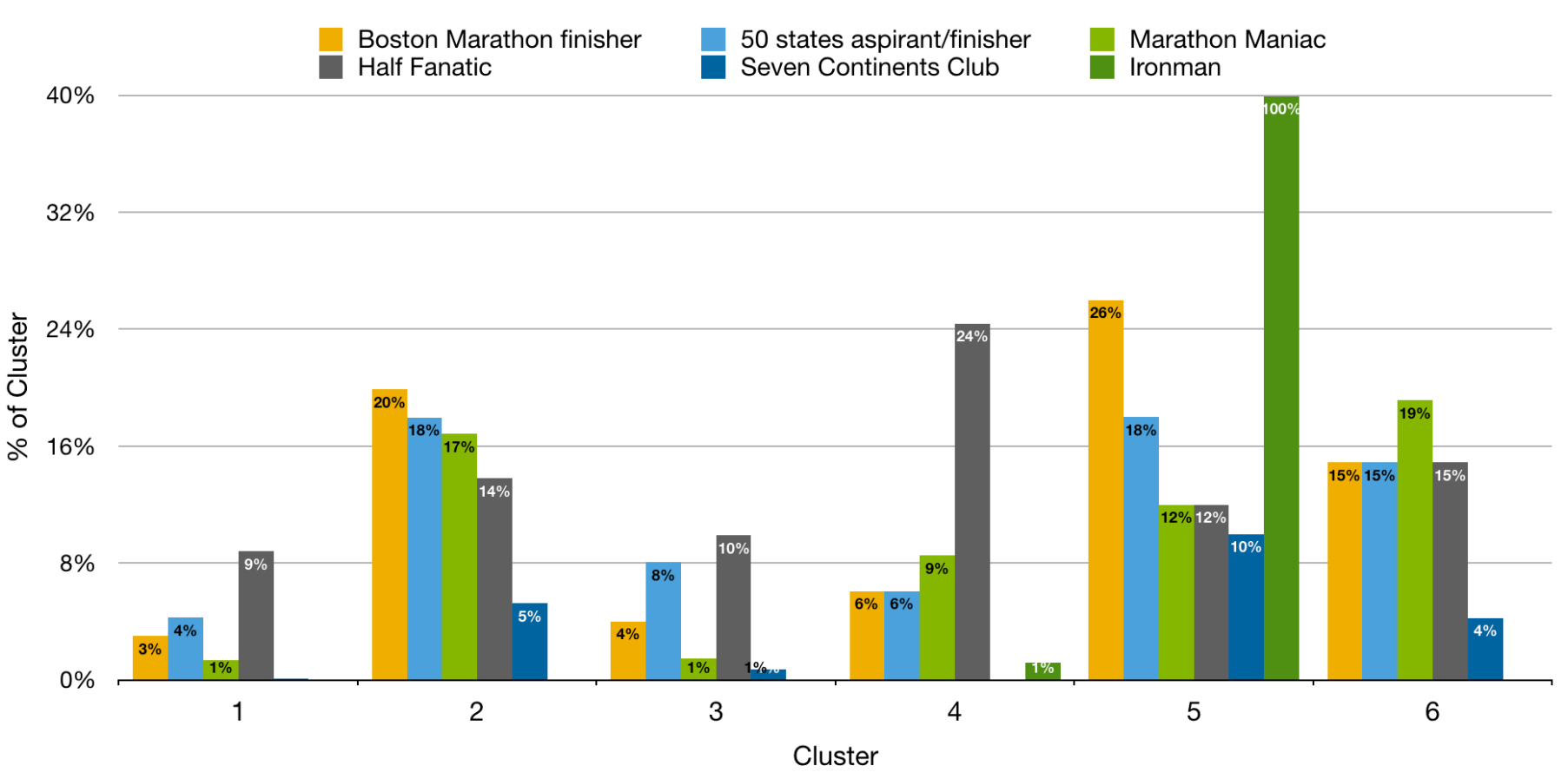





Overall Rating
5
4
3
2
1
0
4.7
4.5
3.3
4.2
4.5
4.3
1
2
3
4
5
6
Cluster


Difficulty
Production
Scenery
Swag
5
4
3
2
1
0
3.3
4.6
4.5
4.4
3.1
4.5
4.1
4.0
2.6
3.0
3.4
2.9
2.8
4.2
3.8
3.8
3.3
4.3
4.3
3.9
3.3
4.2
4.1
3.8
1
2
3
4
5
6
Cluster


Half Marathon
Marathon
10K
5K
10 Miler
100%
80%
60%
40%
20%
0%
% of Cluster
77%
1%
14%
8%
37%
91%
6%
3%
62%
12%
15%
26%
3%
95%
56%
83%
74%
49%
50%
48%
4%
4%
60%
53%
28%
15%
9%
1
2
3
4
5
6
Cluster



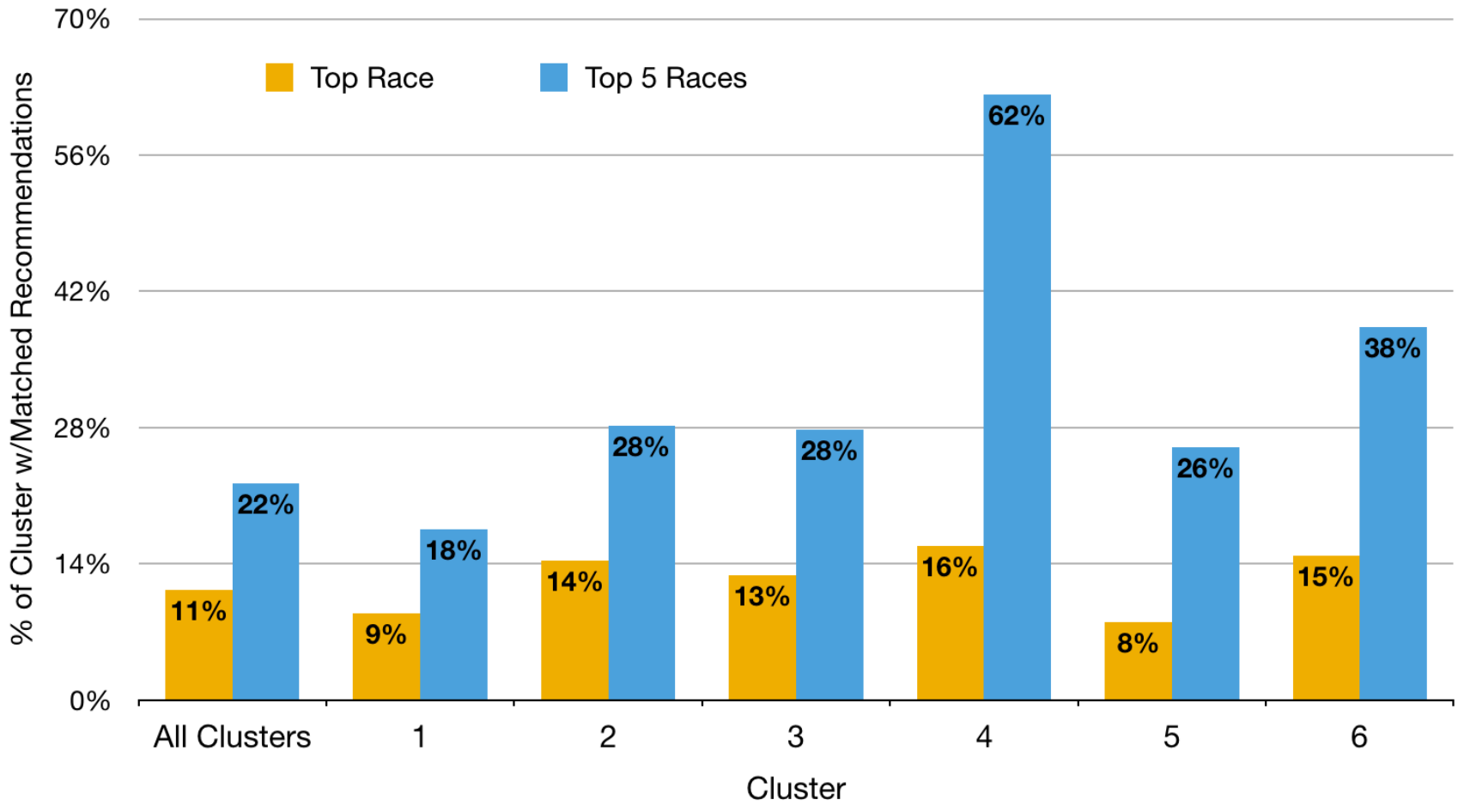
Below is a short summary of each of the clusters based on the charts above:

- Cluster 1: Largest cluster with the fewest average reviews but highest ratings; mostly half marathon runners
- Cluster 2: Marathon runners; least likely to review half marathons; most Boston Marathon finishers are here
- Cluster 3: Lowest ratings, mostly half marathon runners
- Cluster 4: Most engaged users, with an average of 19 reviews and the most distances reviewed
- Cluster 5: 100% Ironman affiliation, equally as likely to have reviewed marathons and half marathons
- Cluster 6: Second highest average # of reviews, but lower ratings and higher affiliations than Cluster 4

Clusters and Race Recommendations

Now that we have the racer population broken down into clusters, we can look at the hit rates for each of the clusters. Not surprisingly, Cluster 4 has the highest hit rates by far. This is because they wrote the most reviews, so we have richer data and a higher probability of matching in the first place. Still, there were 1,768 races to assign, and the recommender found matched races for 62% of the racers in this cluater.

Hit rates are lowest for Cluster 1, which had the fewest average reviews. This indicates that increasing engagement across users of the site would improve recommender results.



Next Steps

There are several ways that this recommender can be enhanced:

- First and foremost, the wesbite already includes some filters in the Find a Race function. We need to combine the filters with the recommender to see if the resulting races make even more sense.
- The recommender was built without any consideration for ratings. The next phase of the recommender build should incorporate ratings to ensure that recommended races meet a threshold. Ultimately, this should incorporate all five of the rating categories.
- The owners of RaceRaves.com have indicated that they would like to add an indicator of how the user was sourced to the clusters/profiles, to understand if users who got a promotion behave differently.

In [ ]:

In [2]:

RaceRaves.com Race Recommender

NLP-based recommender engine built using race reviews from the RaceRaves.com website

Problem Statement:

Runners are always looking for new challenges and setting new goals for themselves. But it's not always easy to find that next race that will help you make that goal. The information available online is scattered and inconsistent. RaceRaves.com does a great job of collating race information and collecting reviews from racers. They also have a great Find A Race feature, but it really just tells you which races are coming up that meet the criteria you select. And those criteria are currently limited to Distance, Terrain, Geography and Date:

https://raceraves.com/find-a-race/ (https://raceraves.com/find-a-race/)

Once you have the results, you can sort them by date, overall rating or alphabetical. But there may be other factors that determine which race you want to run. For example, you could be looking for a flat course or a scenic route. While RaceRaves could try to anticipate all the factors that might matter to a user, another option is mining the review data to determine what factors matter most to each racer. And once you've established that, you can look for races with reviews that talk about those same features. That is the goal of the recommender engine.

Data:

All of the data for this project was scraped from the RaceRaves website. I began by scraping all races dated from June 2017 through May 2018 using the Find a Race page and setting monthly criteria. I then compiled the user IDs from the race reviews posted to these races and pulled the individual racer data for all users collected from the race pages. The final counts were as follows:

- Unique Racers = 2,009
- Unique Races = 2,103
- Total Reviews = 6,414

All of the post-scraping code for this project can be found here. (https://git.generalassemb.ly/dwaynejarrell/DSI-Capstone/blob/master/Capstone%20Recommender%20Final.ipynb)

Data Cleaning

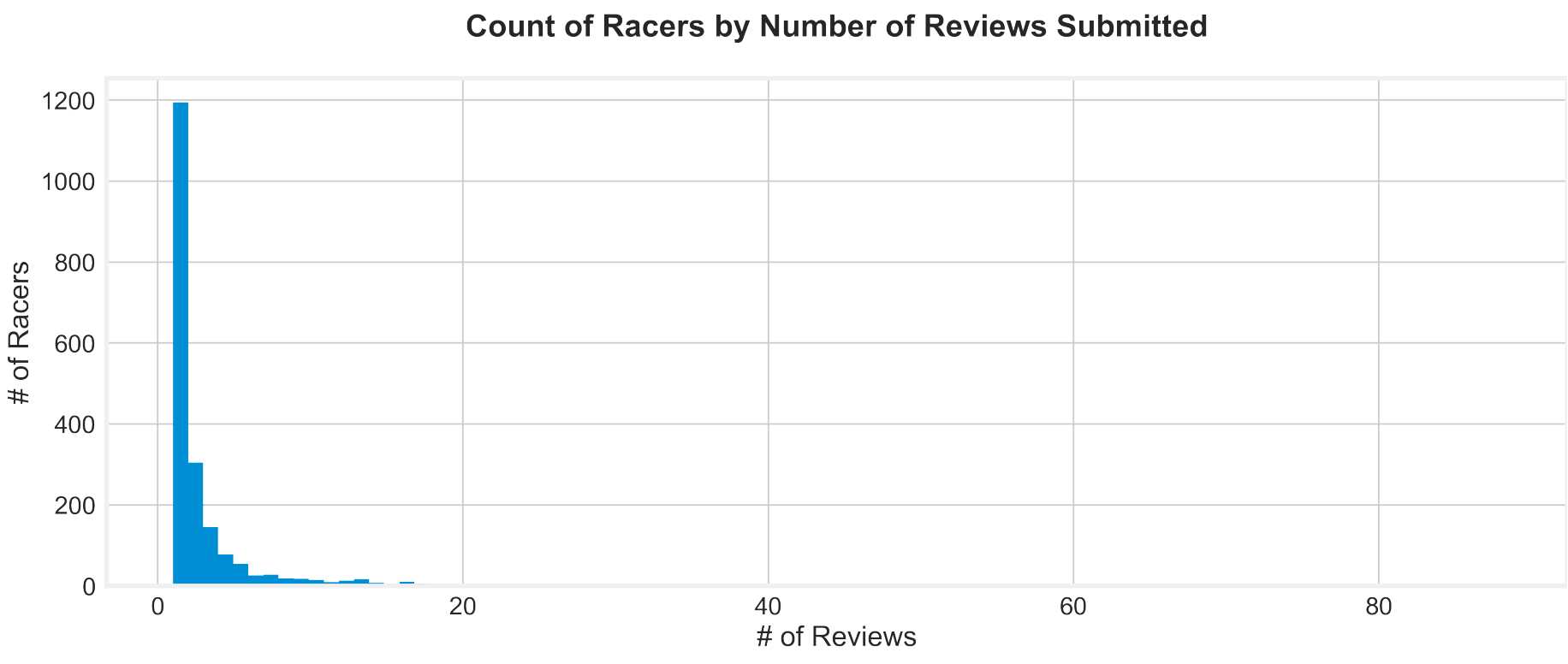
Data cleansing and formatting was built into the scraping process, so there wasn't a lot of cleaning to do once the data was in Python. One exception was the Affiliations feature, which had some old values that didn't correspond to the current options available on the website. For example, some racers had an affiliation of "Ironman athlete", but the current option on the website is just "Ironman". I created a function to update the three old values.

Because the core data came from racer pages, I started out with one row per RaceRaves user ID. Each row could have multiple race reviews, which were stored in a dictionary when scraped from the website. So the first major step was splitting out the reviews into individual rows. I did this by parsing the dictionaries and creating a unique row for each of the 6,414 racer/race combinations with reviews.

After splitting out the data for each review, I noticed that the number of race distances represented in the data was a bit unwieldy - there were a total of 135 different distances, with 58 of those having only 1 review. I decided to limit the the distances to those with at least 20 reviews. That gave me 15 distance categories. All other distances were lumped into 'Other'.

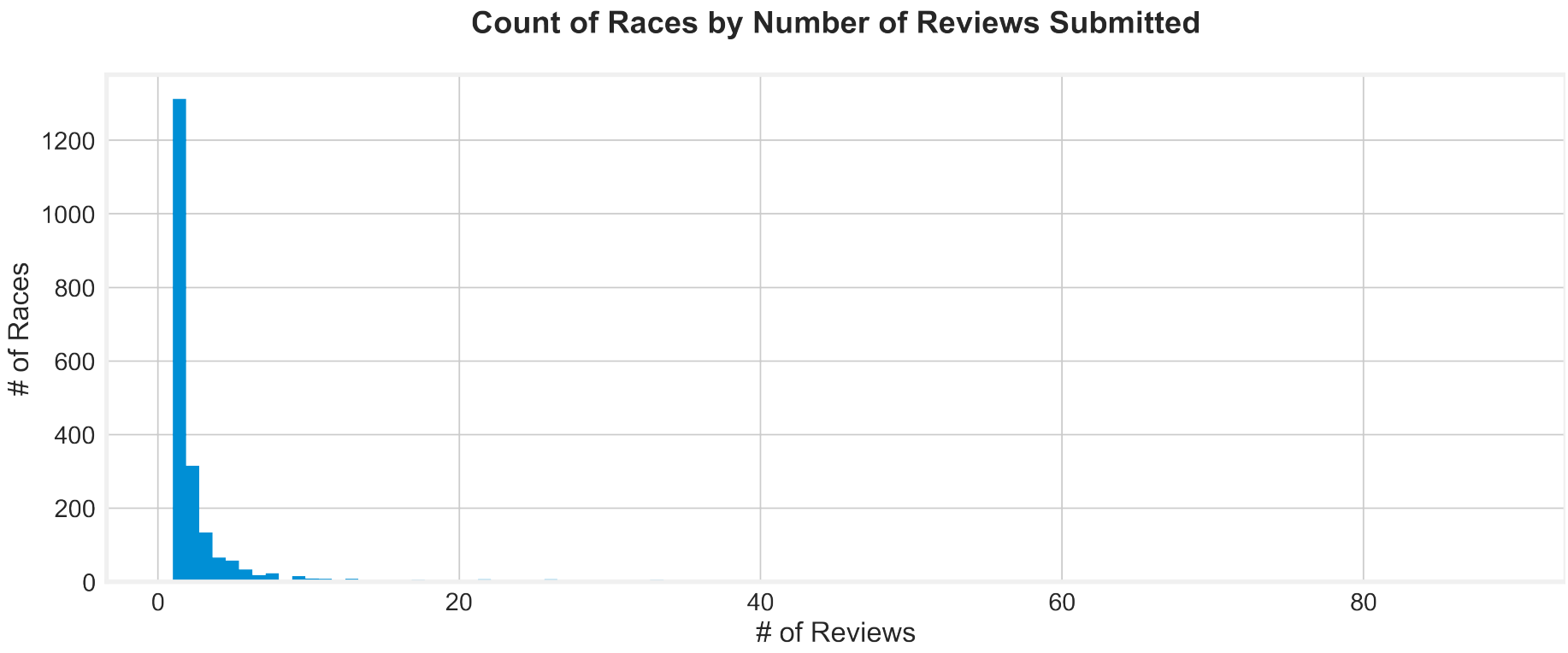
Exploratory Data Analysis

The first step was to get a sense of review frequencies for both racers and racers. Given the nature of the data collection and compilation, each racer was guaranteed to have at least one review. A quick check told me that the average number of reviews was 3.2, but the median was 1 and the max was 88. So we clearly have a skewed distribution, as you would expect:

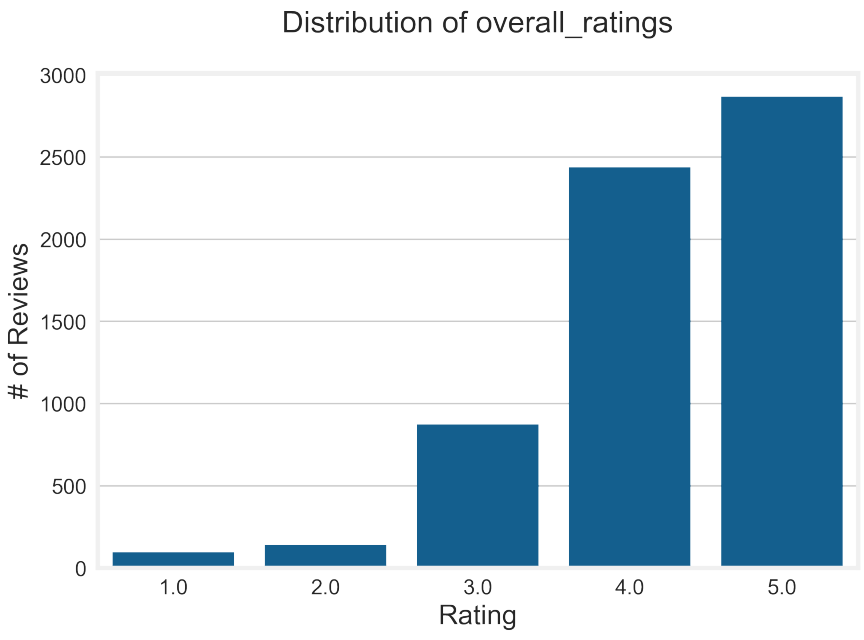


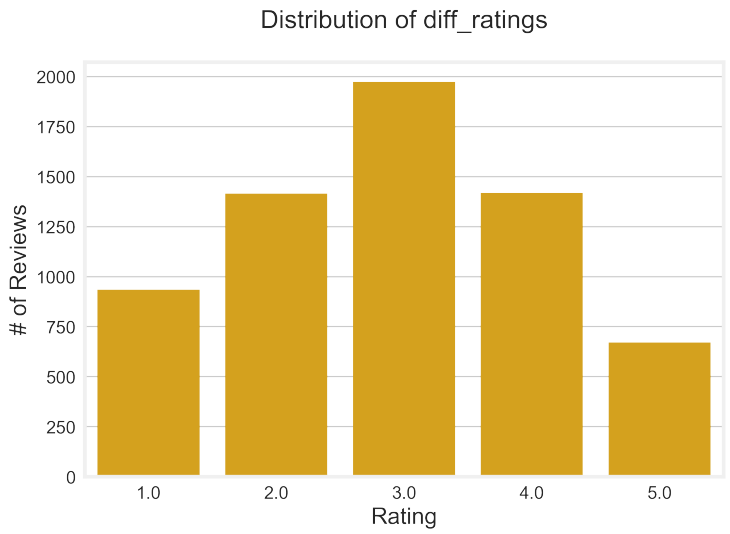
Unfortunately, 59% of the racers (1,195 of 2,009) have only 1 review, but even that one review should give us information on what matters to the racer.

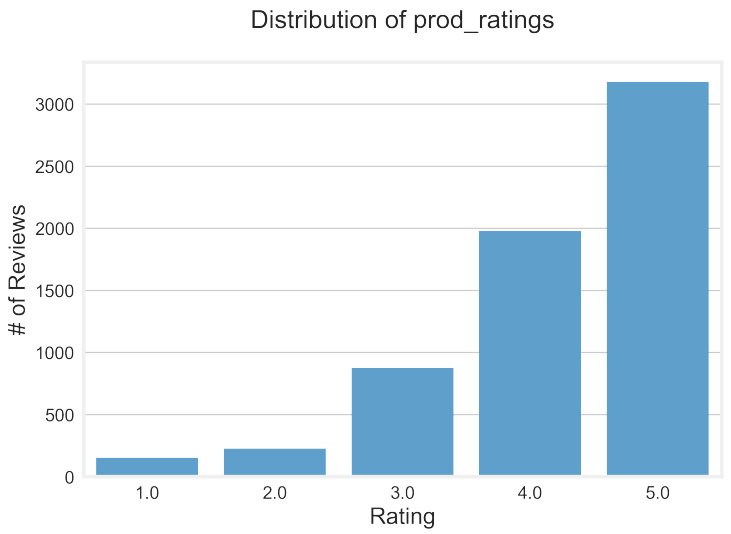
The story for races is similar - 62% (1,131 of 2,102) have only 1 review. Average number of reviews per race is 3.05, and the max is 89.

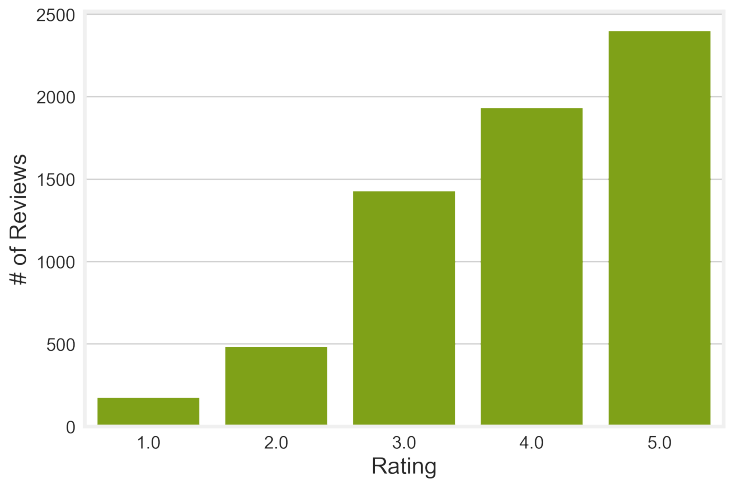


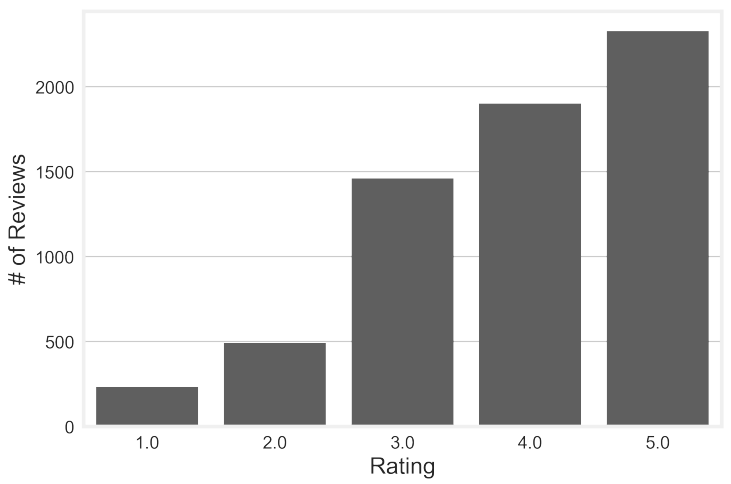
Next, we can look at the ratings. There are five total for every review: Overall, Difficulty, Production, Scenery and Swag. There are some clear differences in the distributions.





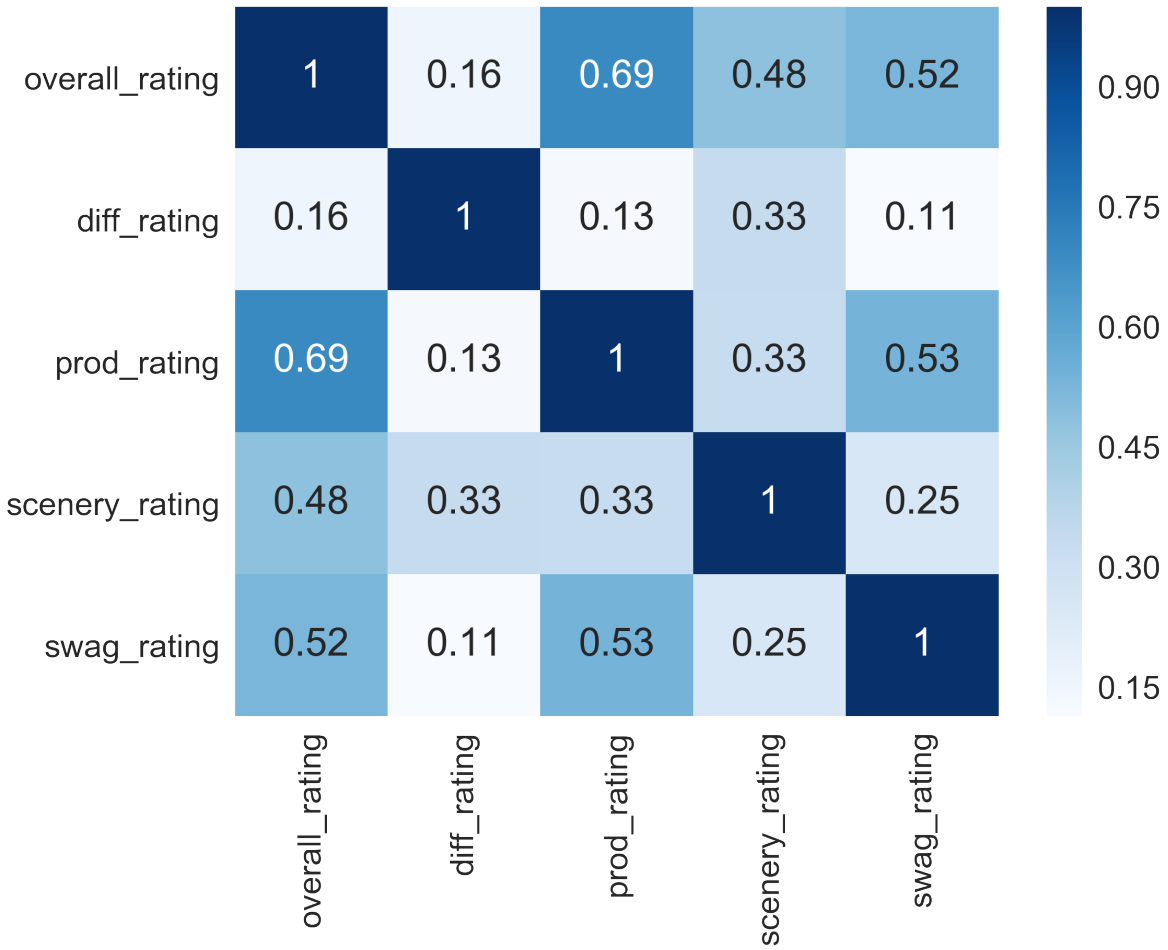






Overall Rating and Production Rating both have means of 4.2, but Production is more likely to be rated a 5 - 50% of reviews gave a production rating of 5, but only 45% of overall ratings were a 5. Scenery and swag are more likely to get rated a 3 than overall and production, but they still get 5 ratings more than any other value. Difficulty, on the other hand, is unlikely to be rated 5 - the average difficulty rating is only 2.9.

Next step was a look at the correlation matrix for all of the ratings:



Clearly, production ratings are the most highly correlated with overall ratings, but scenery and swag are also correlated at 0.5. Difficulty ratings have a very weak relationship with the overall ratings. If we run a simple linear regression to predict overall ratings with production ratings, we get an R-squared of 48%. So we can say that production ratings are driving about half of the variance in overall ratings. If we add the other three ratings to the regression, R-squared only goes up to 58%.

Feature Engineering

Because I am using NLP to build the recommender, feature engineering was focused entirely on the processing of the words in the reviews. This can be broken down into three key parts: n-gram selection, stop words, and stemming.

N-gram selection

Based partly on prior examples of recommenders built from reviews and partly on exploration of the review data for this project, I decided to focus exclusively on bi-grams for all NLP analysis. In particular, the patterns and frequencies of bi-grams corresponded more directly to the themes a runner might include in writing reviews than single words did.

Stop words

After several iterations of running the count vectorizer on the raw review data, I chose to add the following to the standard set of Engligh words provided in sci-kit learn:

- marathon
- because
- mile
- join
- ultra
- ive
- takes
- all numbers (generally used to denote distance, which is captured elsewhere in the data)

Stemming

Instead of stemming words with a standard tool, I did my own analysis of similar words that represent the same basic term or concept. For example, while the term "aid stations" was clearly the most common bi-gram across all reviews, some people used the singular "aid station" or called them "water stations" or "water stops". There were also mutltiple versions of "start" and "run". To address this, I set up a dictionary to map replacement words.

Modeling

Validation

In order to provide some validation of the races recommended to RaceRaves users, I chose to create a holdout sample of 20% of users. The reviews for these users were excluded from the training so I could apply the recommender to them and assess the hit rate for races they've already reviewed. Thus, the first step of modeling was to randomly select the 80% of users who would be used to train the recommender.

Counts for the training vs. test data were as follows:

Training

- Unique Racers = 1,607
- Unique Races = 1,768
- Total Reviews = 5,057

Holdout/Validation

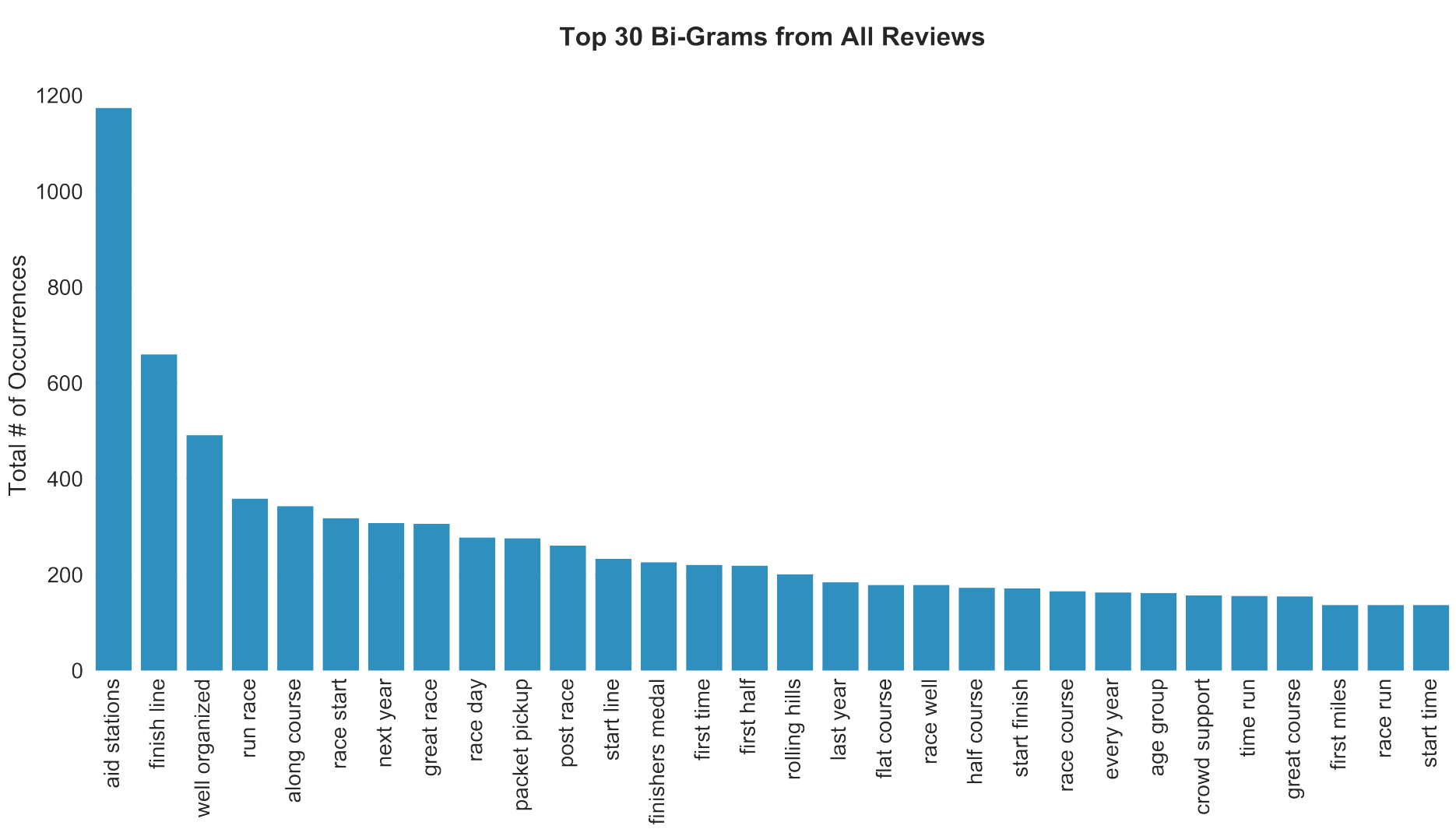
- Unique Racers = 401
- Unique Races = 776
- Total Reviews = 1,357

Count Vectorizer

The first step in the build process was the count vectorizer. Because LDA is expecting word counts, I went with the standard CountVectorizer in sci-kit learn. (I tried TF-IDF, but the results were considerably worse.) Final options selected to maximize hit rates were as follows:

- n-grams = 2 (bi-grams only)
- stop words set to custom list created above
- minimum term frequency set to 25
- maximum term percentage set to 0.25

After running the count vectorizer, which gave me 517 bigrams, I compiled the top 30 bigrams to review and validate as being relevant to racers.



Latent Dirichlet Allocation

The next step was topic assignment using the LatentDirichletAllocation module within sci-kit learn. I tried a wide variety of parameters and evaluated them all with the hit rates. The final parameters were as follows:

- Number of topics = 30
- Learning Method = Online variational Bayes method, which uses mini-batches to update the topics
- Learning Decay = 0.8 (default is 0.7)
- Learning Offset = 10 (default)
- Maximum # of Iterations = 100

In addition to checking the hit rates for training and test data, I reviewed the bi-grams across the topics to ensure that the resulting themes made sense. Here are the top 20 bigrams for each of the 30 topics chosen by the model (in no particular order):

Matching Racers and Races

Assigning Topics to Racers

The first step in matching racers and races was to assign the topics to all of the racers in the training data. In order to do that, I rolled up individual race reviews at the user ID level and created one bag of words from all the reviews for each user. For the purposes of clustering and profiling later, I also collected the number of reviews and number of distances, created dummy variables for each of the distances run and merged in the user's affiliations and average ratings.

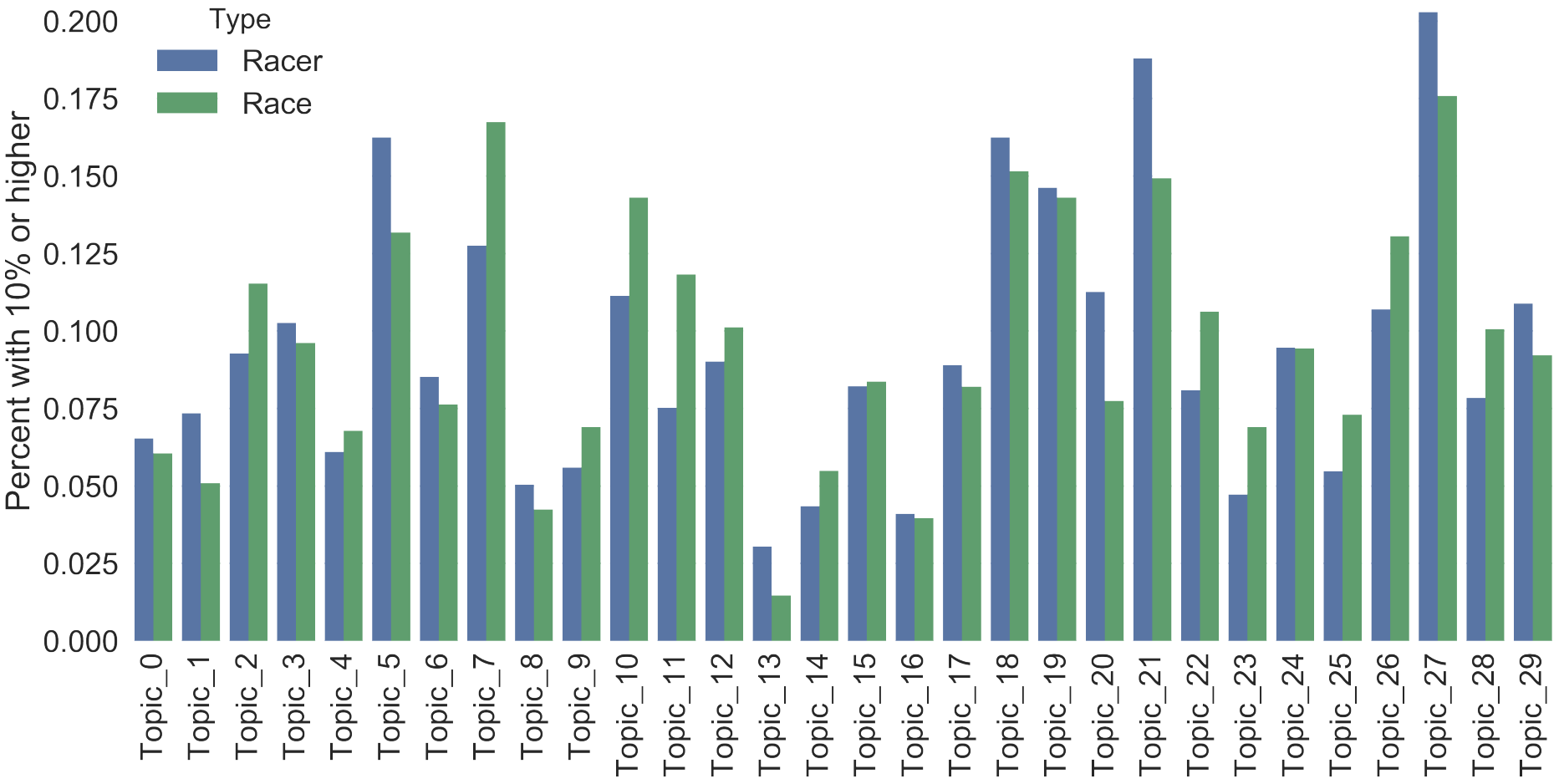
Once I had the racer-level data, I ran the fitted Count Vectorizer and LDA models on the combined reviews. Before fitting the LDA, I divided the counts by the number of reviews to account for the fact that the model was fitted on single reviews. For users with multiple reviews, this means a single bi-gram would be downgraded while a mention of that bi-gram across all reviews would give it full weight in the LDA model.

Assigning Topics to Races

Next step was assigning the topics to all of the races in the training data. This followed the same basic process as the racers. I rolled up individual race reviews at the race name level and created one bag of words from all the reviews for each race. I also collected the number of reviews and distances, created dummy variables for each of the distances available in the race and merged in average ratings.

As I did with the racer data, I ran the fitted Count Vectorizer and LDA models on the combined race reviews. I also divided the counts by the number of reviews to account for the fact that the model was fitted on single reviews, just as I did for racers.

Here's a look at how the topics fell out by Race and Racer, using a 10% threshold to determine topic assignment:



Matching Racers and Races

For each of the 1,607 racers, I calculated the differences in topics between the racer and all 1,768 races using Euclidean Distance. Euclidean distance is the straight-line distance between all 30 topic probabilities, calculated using the Pythagorean Theorem (square root of the sum of squares). I then ranked all races for each user and chose the top 5 recommended races

Hit Rates

Training Data

In order to measure the efficacy of my race recommender, I let the recommender choose races that the racer may have already run and reviewed. From a practical perspective, this makes sense - if a racer liked a given race, he or she is likely to want to run that race again. They would likely be happy to see the race show up as an option, but this is a choice that RaceRaves and/or the user could make. Including these races also allowed me to calculate a Hit Rate, which is the % of times the recommended race is one the person has already run and reviewed.

The hit rates for the training data are below:

- Number of matches for Top Race = **182** (**11.3%**)
- Number of matches Top 5 Races = **358** (**22.3%**)

Note that these hit rates are biased upward by the fact that the same reviews were used to build the recommender and score the races. We need the validation set to truly assess the success of the recommender.

Validation Data

To validate the recommender, I ran the same process on the 20% holdout racers that I did on the training racers - roll up reviews to racer level, apply the count vectorizer, assign topics using LDA, and then calculate distance and rank races. It is important to note that the set of races I matched to (the same 1,768 used above) contained NO REVIEWS from these RaceRaves users - all of their reviews were held out during the initial random selection. So any matches we find are based on matching these users' review topics to the topics from other users' reviews.

Here are the results:

- Number of matches for Top Race = **9** (**2.2%**)
- Number of matches for Top 5 Races = **29** (**7.2%**)

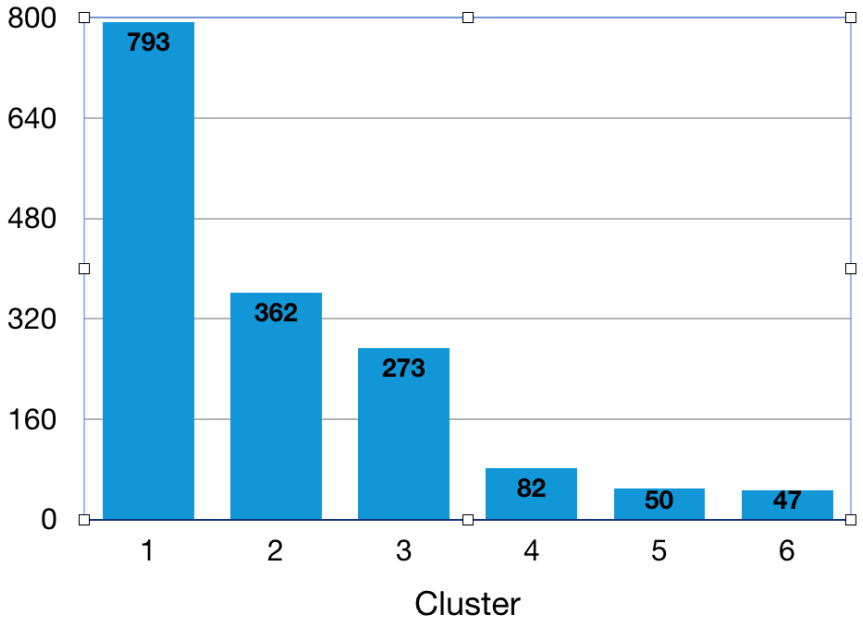
It's also true that excluding the holdout reviews meant that nearly half of the races reviewed by this population (334 out of 776, or 43%) were not included in the training data. There were only 442 races that existed in both the training and holdout data. The true hit rates would be more than double if those races could have been included in the training. All of which indicates that the recommender is doing a good job of matching up racers with appropriate races.

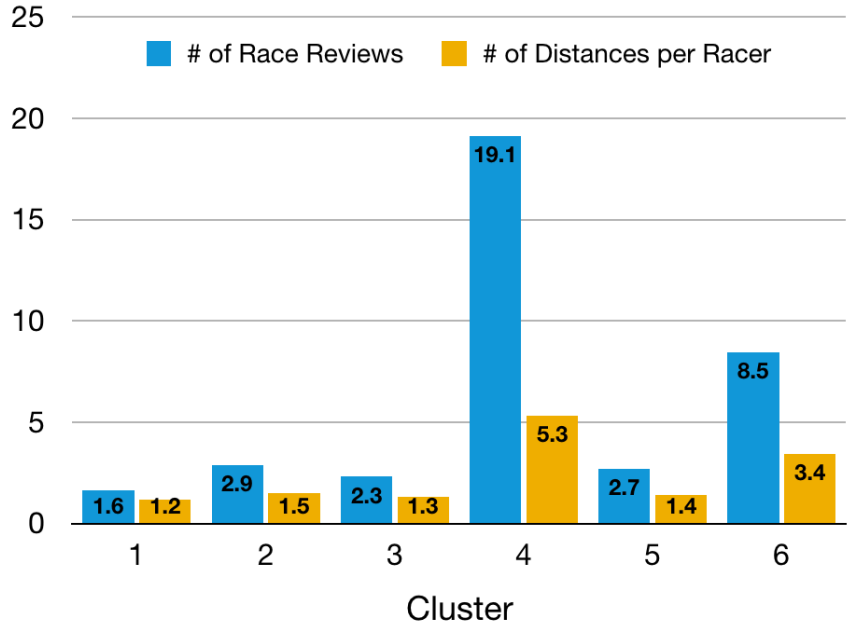
Clustering

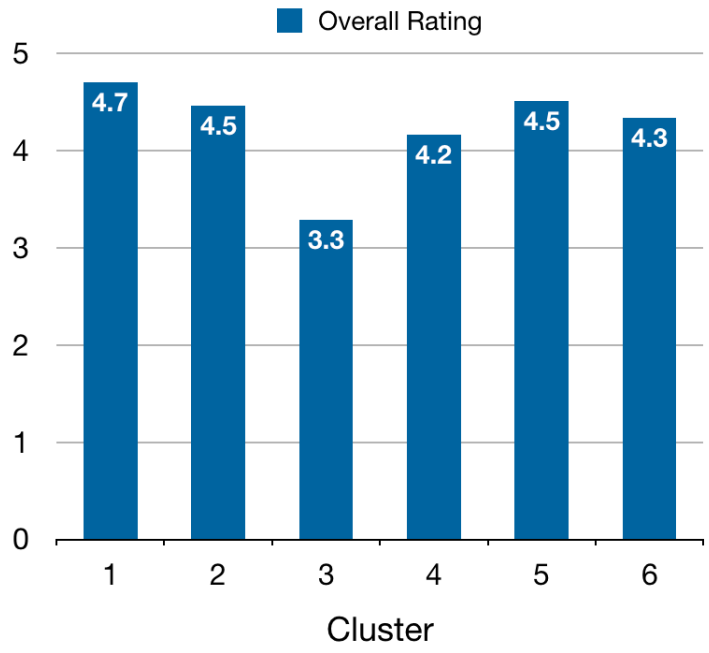
In addition to completing the initial phase of the Race Recommender, I wanted to take a look at the users of RaceRaves.com and understand their review/rating behaviors and see if we can break them out into recognizable clusters. I tried a few methodologies, but the one that gave the best results was K-Means clustering using the following profile features:

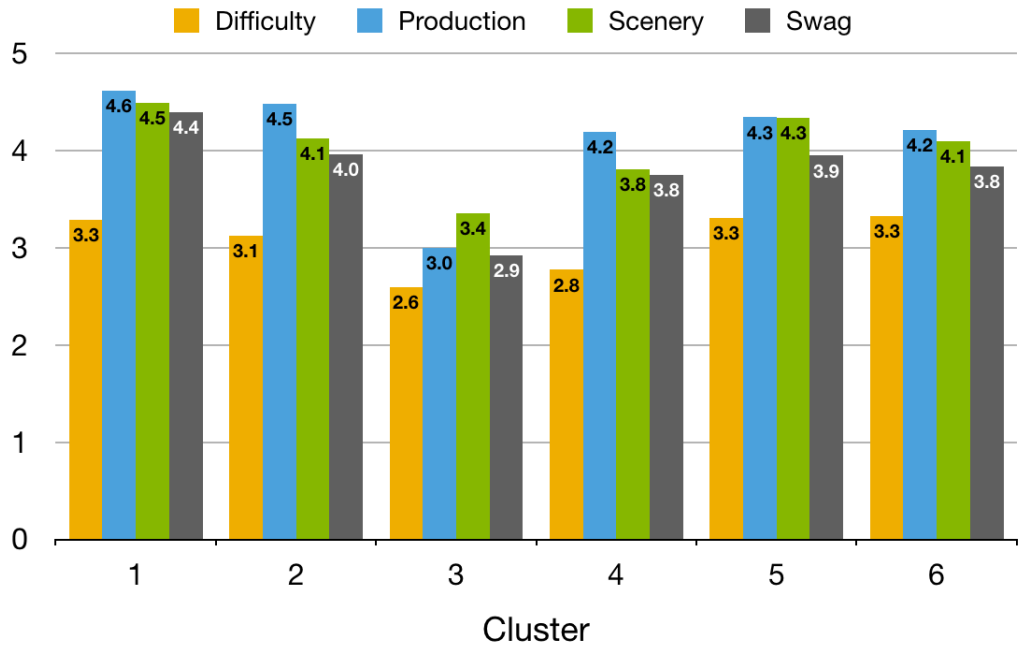
- Number of Reviews
- Number of Distances
- Average Ratings (all 5 factors)
- Distances run (Marathon / Half Marathon / 10K / 5K / 10 Miler / 12K / 15K / 50K / Other)
- Affiliations
- Topic Probabilities

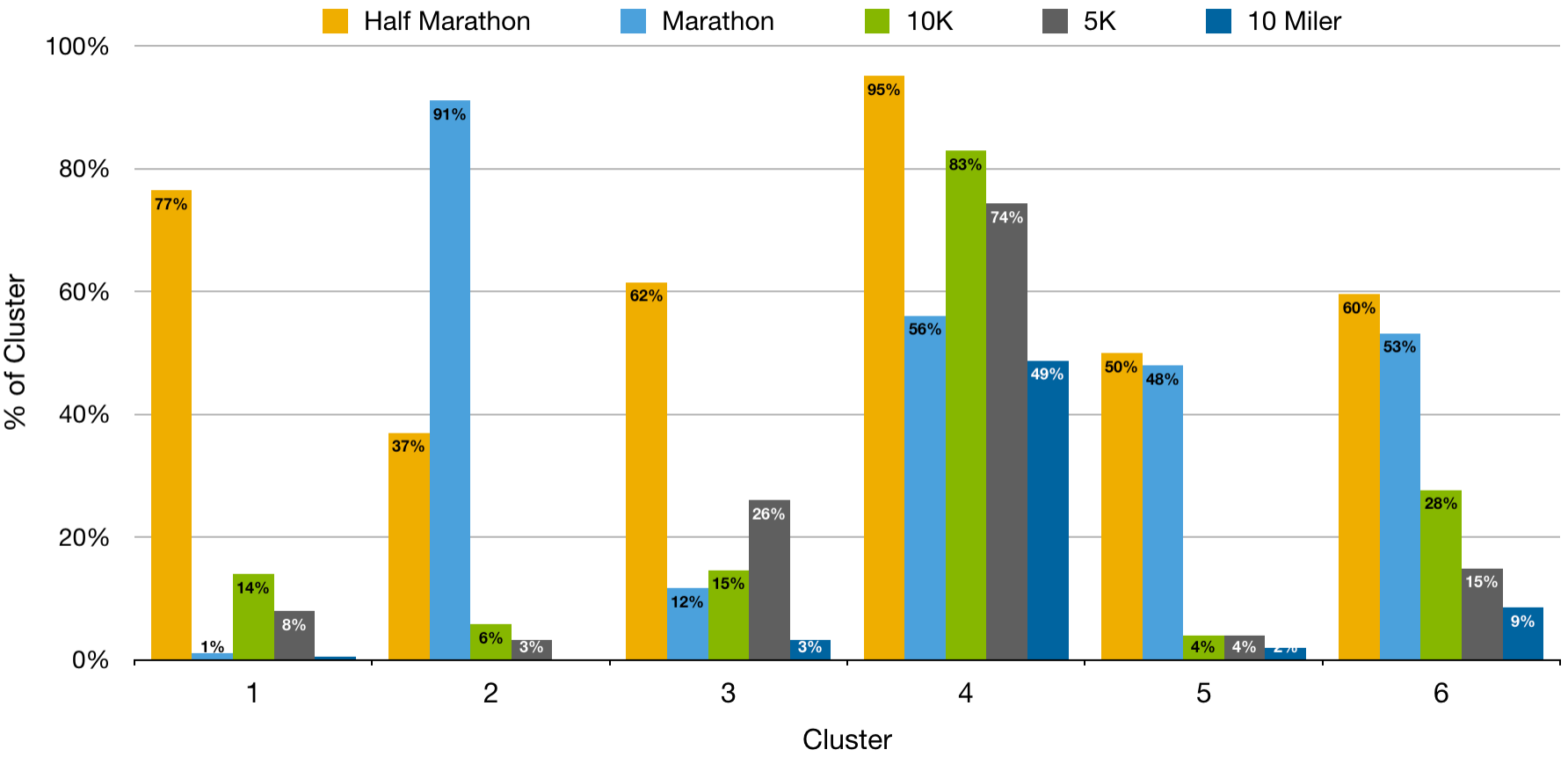
The final specifications I chose, based on the relative size and cohesion of the cluster, were 6 clusters, 25 initial centroids and max iterations of 300. I also standardized all of the features prior to fitting the K-Means. The following charts summarize the resulting clusters.

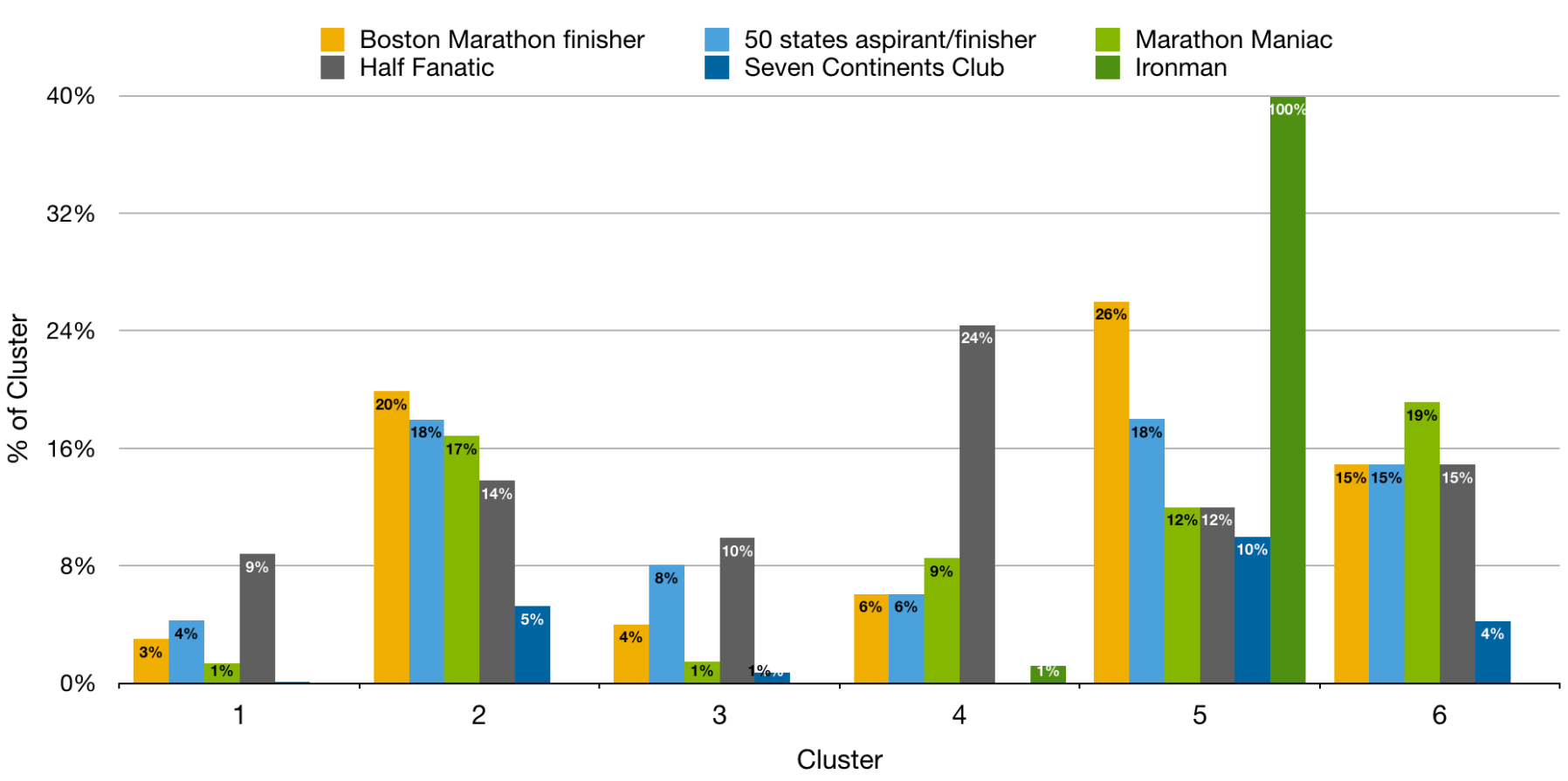





Overall Rating
5
4
3
2
1
0
4.7
4.5
3.3
4.2
4.5
4.3
1
2
3
4
5
6
Cluster


Difficulty
Production
Scenery
Swag
5
4
3
2
1
0
3.3
4.6
4.5
4.4
3.1
4.5
4.1
4.0
2.6
3.0
3.4
2.9
2.8
4.2
3.8
3.8
3.3
4.3
4.3
3.9
3.3
4.2
4.1
3.8
1
2
3
4
5
6
Cluster


Half Marathon
Marathon
10K
5K
10 Miler
100%
80%
60%
40%
20%
0%
% of Cluster
77%
1%
14%
8%
37%
91%
6%
3%
62%
12%
15%
26%
3%
95%
56%
83%
74%
49%
50%
48%
4%
4%
60%
53%
28%
15%
9%
1
2
3
4
5
6
Cluster



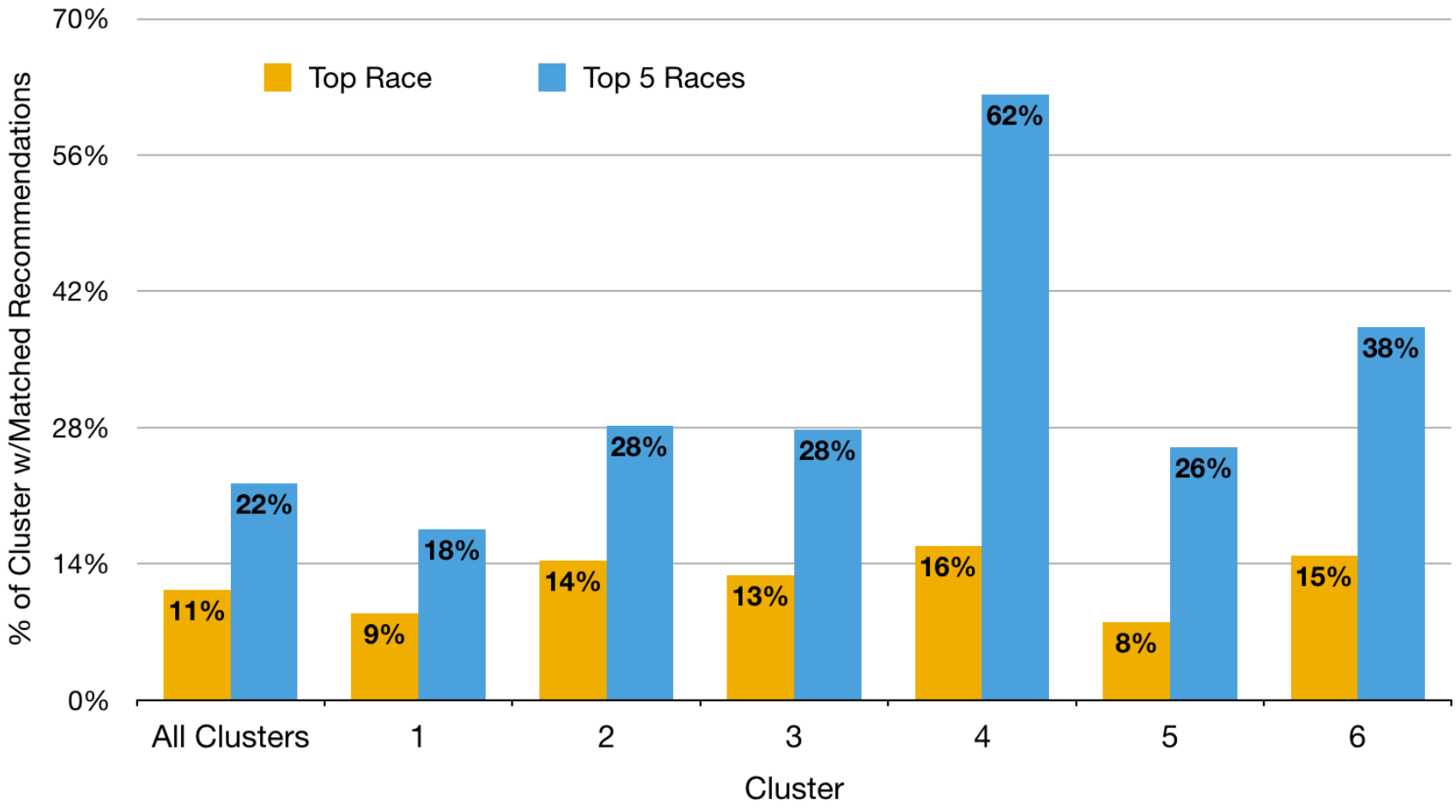
Below is a short summary of each of the clusters based on the charts above:

- Cluster 1: Largest cluster with the fewest average reviews but highest ratings; mostly half marathon runners
- Cluster 2: Marathon runners; least likely to review half marathons; most Boston Marathon finishers are here
- Cluster 3: Lowest ratings, mostly half marathon runners
- Cluster 4: Most engaged users, with an average of 19 reviews and the most distances reviewed
- Cluster 5: 100% Ironman affiliation, equally as likely to have reviewed marathons and half marathons
- Cluster 6: Second highest average # of reviews, but lower ratings and higher affiliations than Cluster 4

Clusters and Race Recommendations

Now that we have the racer population broken down into clusters, we can look at the hit rates for each of the clusters. Not surprisingly, Cluster 4 has the highest hit rates by far. This is because they wrote the most reviews, so we have richer data and a higher probability of matching in the first place. Still, there were 1,768 races to assign, and the recommender found matched races for 62% of the racers in this cluater.

Hit rates are lowest for Cluster 1, which had the fewest average reviews. This indicates that increasing engagement across users of the site would improve recommender results.



Next Steps

There are several ways that this recommender can be enhanced:

- First and foremost, the wesbite already includes some filters in the Find a Race function. We need to combine the filters with the recommender to see if the resulting races make even more sense.
- The recommender was built without any consideration for ratings. The next phase of the recommender build should incorporate ratings to ensure that recommended races meet a threshold. Ultimately, this should incorporate all five of the rating categories.
- The owners of RaceRaves.com have indicated that they would like to add an indicator of how the user was sourced to the clusters/profiles, to understand if users who got a promotion behave differently.

In [ ]: