THOMPSON RIVERS UNIVERSITY

COMP 4610

ADVANCED DATABASES

Big Dota Reduction: Analyzing Over One Billion Dota 2 Matches

Author	*Author*	*Supervisor*
Marco LUSSETTI	Dyson FRASER	Dr. Mila KWIATKOWSKA

April 15, 2019

Note #1: this report is based on our work as submitted in the course of our previous submissions during the course as well as our submission to the TRU Undergraduate Conference. We have omitted direct citations to the submission outside of this paragraph for convenience purposes and as they form the same body of work (Lussetti & Fraser, 2019).

Note #2: the full source code for this report, as well as for the poster, and the implementation of utilities & analysis necessary for this project are available on GitHub at marcolussetti/opendotadump-tools. This includes a README pointing to the key files and folders in the repository.

1 Introduction

The availability of large data sets of player choices in popular online computer games presents an opportunity to identify sudden changes in choice patterns and explore what factors may contribute to such changes. This would be a form of time-series analysis, with the seasonality determined at least partially by meta-game events rather than exclusively the Gregorian calendar. It is thought that industry is ahead of academia in working with very large data sets (Jin, Wah, Cheng, & Wang, 2015) and we hope that by showcasing the feasibility of poor man's solutions to very large data sets we can help encourage academics with limited resources to undertake big data work.

We are particularly interested in exploring one of the largest publicly available data sets, a data set from the OpenDota Project (The OpenDota Project, 2014) containing all matches played in the online video game Dota 2 over five years due to its large size and detail (The OpenDota Project, 2017). We are interested in analyzing the big data reduction techniques required to handle such data sets, as well as the engineering challenges involved in processing a data set that cannot be stored in memory in its initial form and thus must be handled via stream processing. Of great interest to us are approaches and techniques that are viable with the limited computing resources that may be available to individual researches or small teams.

1.1 What is Dota 2?

Dota 2 is a online multiplayer game that has two teams of 5 players each competing to destroy each others Ancient, a large structure located in the

center of their base. To win the game one team's ancient must be destroyed but there are many obstacles that stop teams from rushing in and destroying it in the opening minutes of a match. There is a large jungle separating the two bases with three paths, or lanes as they are more commonly called, leading from one base to the other. The half of the lane is part of one team's territory and the other half belongs to the other team. Turrets are stationed on each teams half in order to hinder progress to the Ancient.

Players choose from a list of playable characters call Heroes at the start of each match and level them up by killing creeps (Non Playable Characters spawned from the Ancient), other non-playing characters that are located in the jungle and enemy heroes. The experience points that players gain by playing a Hero does not transfer to another match. Heroes have been designed for a specific role that causes them to mainly be played in one of these lanes. For example there is a type of character called a Carry that is very frail but can do lots of damage, this type of character in combination with a Support(another role) play together in the same lane. Hero picks are a very important part of the game and I would now like to explain the mechanics behind hero selection. At the start of each game, players pick from the pool of heroes, 115 heroes exist but at the time our data was collected only 111 were released, once one player on a team has picked a hero no other player on that team can pick that hero for that match. The result after picking heroes is a unique set of characters that must be leveled up over the course of the match. Players often migrate to a specific role such as carry or support and only play characters within that select role as that is what is most enjoyable for them. So their Hero pick pool is shrunk from 111 to 30. Within this group of 30 heroes players must now decided who to play and there are many things to take into account when choosing a Hero most importantly is how good that character is.

The state of viability for a hero varies as time passes. If a hero is picked too much or their win ratio is too high Valve, the developers of Dota 2, will update the game and implement changes that will make the hero worse. This is known as a Nerf, an update to a over powered item or character in a game that takes away some of its power in order to balance the other items and characters of the game. Inversely, if a character or item is under powered a Buff can take place which makes the item or character better in hopes of raising its win ratio or pick rate to similar levels as its contemporaries. Nerfs and buffs are both implemented in the game in the form of updates more commonly known as Patches. Based on the current power level of a Hero, it may become a more desirable pick among players. Another thing that may sway a player's choice would be the implementation

of a new Hero that fulfills a role that the player enjoys to play. These new heroes are also introduced via patches.

1.2 Research Objectives

- Detect metagame shifts from hero pick ratios, that would be caused by external events (patches to the game, major tournaments, etc.)
- "Tame" the dataset's enormity using the simplest tools possible, and using only the resource of a normal machine as may be available to an average researcher without extensive funding

2 Methodology

2.1 Materials

Our research relied on the Open Dota Data Dump, a data dump published by the Open Dota project in 2017. This is a collection of meta data on 1.2 billion matches over five years (The OpenDota Project, 2017). It consists of three files extracted from the same collection of meta data. *match_skill* provides information on the Valve-estimated skill level of the match, *matches* provides meta data about all matches in the data set, and *player_matches* provides detailed in-match data about a subset of matches. Our research will use the *matches* file as we are only interested in basic meta data about each match, and were interested in having as matches to work on as possible. It should be noted that while sample files exist for all these files, the *player_matches* file is no longer accessible from its original torrent due to lack of seeders.

The *matches* file is a comma-separated value file with 27 columns, one of which, *pgroup* is an embedded JSON fields. Figure 1 shows the attributes of this file. The asterisk denotes the columns we have identified as possibly needed for our analysis.

2.2 Methods

2.2.1 Big Data Reduction In Theory

Because of the large size of the data set, we quickly determined that we would need to not only employ dimensionality reduction to reduce its size before analyzing, but to also aggregate it at some level.

Figure 1: Fields in *matches* file

- *match_id*
- *match_seq_num*
- *radiant_win**
- *start_time**
- *duration*
- *tower_status_radiant*
- *tower_status_dire*
- *barracks_status_radiant*
- *barracks_status_dire*
- *cluster*
- *first_blood_time*
- *lobby_type*
- *human_players*
- *leagueid*
- *positive_votes*
- *negative_votes*
- *game_mode*
- *engine*
- *picks_bans*
- *parse_status*
- *chat*
- *objectives*
- *radiant_gold_adv*
- *radiant_xp_adv*
- *teamfights*
- *version*
- *pgroup**

The issues associated with high dimensionality data sets are well understood, and in our context chiefly relate to the increased storage requirement and computational complexity required to handle such data set. In our case, the sparsity issues that might be associated with high dimensionality are not a real concern for our study because of the large size of the data set itself and the fact that we are not using the data set to make prediction relying on these dimensions (Pore, 2017; ur Rehman et al., 2016). In terms of dimensionality reduction thus, we were able to condense our data set to merely the heroes picked, the time of the match, and which team won.

However, dimensionality reduction alone would not condense the data set significantly enough to allow for processing on our limited resources.

We established this by manually creating a dimensionality reduced-version of 637309 lines, the average lines per day. While this benchmark is admittedly crude as all lines are identical, it did give us some idea of what we could achieve via dimensionality reduction alone. One day represented in this manner amounted to 94.8 MB, which compared to the original dimension of 670.56 MB ($1.12TB/1870days$) represented a compression of approximately 7 times. Thus the complete data set would be estimated at around 160GB, a significant reduction but alas not enough.

Our next step in condensing the data is to examine whether any viable option exists for granularity reduction. The concept of an information granule is used in granular computing to indicate the minimum unit employed in the model. Thus, the larger the granule can be thought of as being, the more information it abstracts away (Yao, 2004). In our case, we are attempting to compare days rather than individual matches, and thus can aggregate hero picks on a day by day basis. We can think of our information granule as moving from being an individual match to being a day of matches treated as an individual unit. In our benchmarking, again using repeated entries, we estimated that decreasing the granularity of the data would yield a 2.73 KB per day file which would be an approximately 34600 times reduction.

2.2.2 Big Data Reduction Implementation

We implemented both dimensionality and granularity reduction jointly as a Java utility. We chose Java as a performant language with a large selection of libraries.

We need to keep in mind that the data set cannot be loaded into memory at once and thus must be treated as a form of stream. However, we do have the advantage of the stream being bounded, and thus may do without a real-time stream processing stream. Stream processing is well understood and is normally tackled by solutions such as cloud computing (Iyer, Sood, Gupta, & Panda, 2013) or tools like Hadoop or Spark (Namiot, 2015). However, because we intended to avoid deploying significant computing resources as would be needed to set up a cluster, we developed our own *poor man's solution* to process the data set.

We originally intended to and experimented with using Java 8 Streams to avoid having to decompress the match data set and adopt a more modern interface, however we quickly determined that the streams introduced significant overhead and were not well suited for the size of data we needed to process.

The input file we are seeking to process is a comma separated value file. We originally investigated the feasibility of developing our own bare-bone parser, however that it was not a viable option due to the complexity involved. We investigated several options for Java CSV parsing libraries. General purpose CSV parsing libraries like *opencsv* (Rucker Jones, Conway, & opencsv Contributors, 2019) do not possess adequate performance (in our early trials we saw performance as low as 1,000 records per second). We reviewed benchmarks for CSV libraries that suggested that the fastest library may be univocity-Parsers (uniVocity Software Pty Ltd, 2018). However when we attempted to implement a univocity-Parser (uniVocity Software Pty Ltd, 2019) based-solution in our utility, we discovered that while the performance claims may well be justified, the library did not cope well with large amounts of records. We found that it would use as much as 16GB of RAM to process only 400,000 records and increasing the allocation of memory to the JVM to 24 GB only allowed an additional 50,000 records to be processed. We thus decided to investigate alternative libraries that might have a thinner memory footprint. We settled on the second fastest library in the benchmark, SimpleFlatMapper (Roger, 2019). This library allowed us very good performance (peaking at 100,000 records per second) while maintaining a low memory footprint. We treated all columns as strings, and only performed parsing on the necessary columns.

The *pgroup* field is an embedded JSON field so we chose to rely on Jsoniter (?, ?) as a JSON parser. The library's performance appeared adequate so we did not investigated the matter significantly. We liked that it did not strongly bound to classes unlike some other libraries thus allowing for faster prototyping.

We also investigated more memory-efficient data structures for holding the resulting data, for which we relied on Trove4J (Eden, Parent, Randall, Friedman, & Trove Development Team, 2013) library which produces some interesting memory efficiency gains for HashMaps (Vorontsov, 2014). However, this proved to be a premature optimization that was wholly unnecessary due to the small size of the HashMaps used in the program.

Last we should note that we relied on picocli (Popma, 2019) for the command line interface implementation.

The Java implementation of this portion is available as *Appendix A:* matches_condenser -> Main.java or on our GitHub repository as the matches_condenser utility.

Some additional conversion tools in the pipeline are implemented in Python and available as *Appendix B:* json_to_csv -> opendota_jsontocsv.py or in our GitHub repository as the json_to_csv tool which includes parts of

the needed data cleansing and processing.

2.2.3 Extraction of Metagame Shifts

To detect the shifts in the metagame, we decided to attempt a very low tech approach. One may imagine each day as a vector of 111 dimensions where each dimension represents the pick rate for a given hero (thus all of those pick rates always add up to 1). We decided to compare each day's vector to a vector representing the running average of the last two weeks (14 days). We used Manhattan distance to compare the vectors.

This produces a comparable value over time, where we may identify prominent peaks as highlighting changes in the metagame. This is well suited to visual identification: the peaks are slowly absorbed over time as they are incorporated into the running average thus creating a peak both in height and width (representing respectively the level of change and the time taken for the new changes to become the normal metagame).

This was implemented as a set of Jupyter Notebook, which also produced the graphs used in this document and the previous presentation. They may be reviewed as *Appendix D:* analysis -> LookupSpikes.ipynb or in our GitHub repository under our analysis tools.

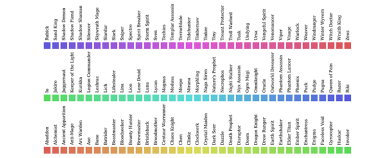
3 Results

In first place, we were able to produce an aggregate chart of the pick ratios, which shows the pick ratios of all champions over time. This is visible in Figure 2, however it is impossible to reproduce the chart with adequate details in this context and we recommend reviewing the full scale chart in its entirety which is referred to in the chart's caption.

From our analysis of the Manhattan distance of each day to the running average of the last two weeks, we were able to produce a representation of metagame change points which may be seen in Figure 3. We have manually highlighted some of the more prominent peaks which we will attempt to identify the causing event for. This can be seen in Figure 4.

As may be seen in the figures, we believe our algorithm was successful in identifying major metagame shifts. Most major peaks could be tracked down to specific patches issued by the developer, and we did not see sizable shifts caused by tournaments such as The International. We did run our analysis with other options such as median instead of average of the pick rates, 28 days average rather than 14 days, and other distance mea-

Figure 2: All Heroes Pick Ratios (full size)



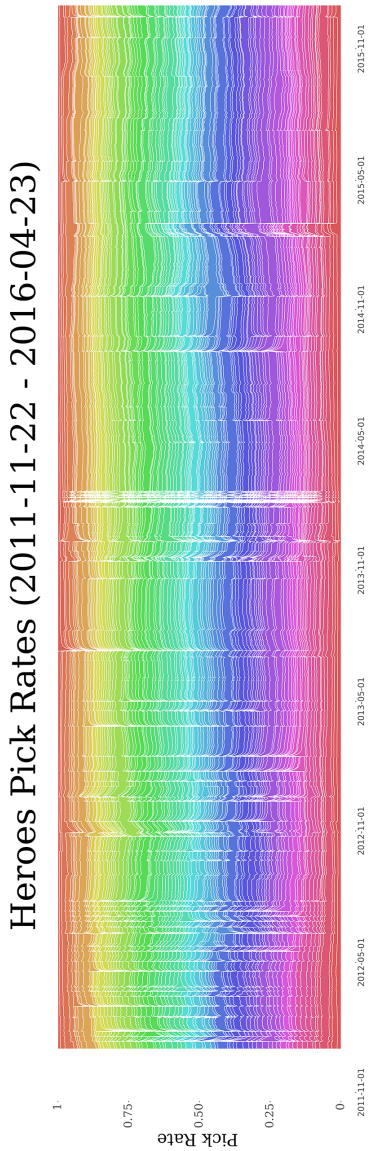


Figure 3: All Heroes Difference (full size)

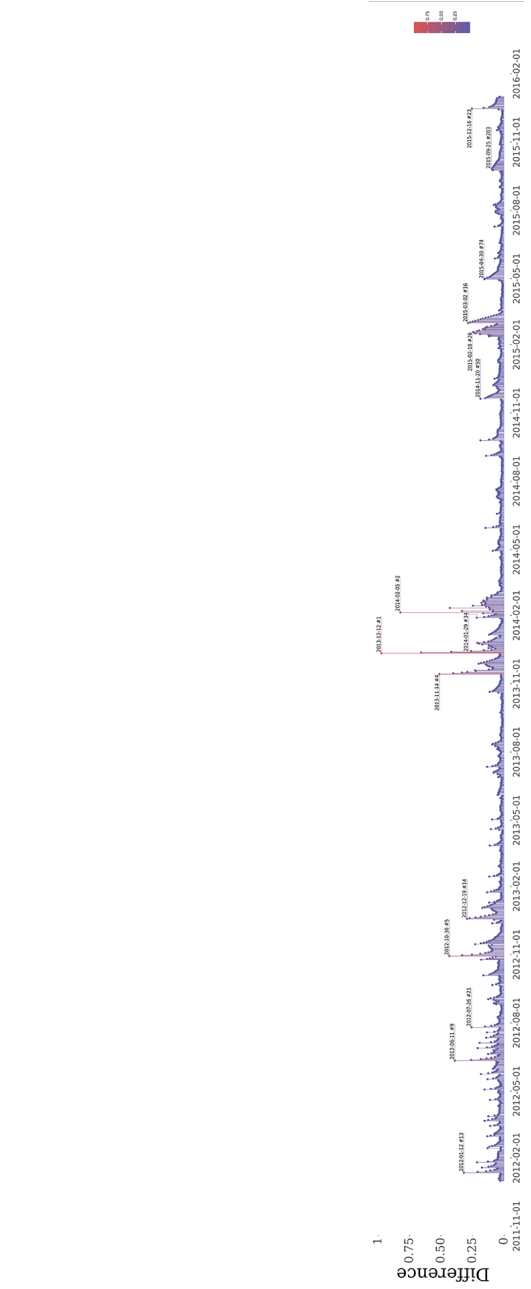


Figure 4: List of major metagame shifts

- *2012-01-12* Major rebalancing of heroes & item changes (Explanatory table for 2012-01-12)
- *2012-06-11* Chaos Knight, Phantom Assassin, Gyrocopter released (Explanatory table for 2012-06-11)
- *2012-07-26* Nyx Assassin, Keeper of the Light, Visage released (Explanatory table for 2012-07-26)
- *2012-10-30* Recently released Centaur Warrunner is nerfed (Explanatory table for 2012-10-30)
- *2012-12-19* Major rebalance of most/all champions (Explanatory table for 2012-12-19)
- *2013-11-14* Three Spirits Patch, significant out-of-game & economy changes (Explanatory table for 2013-11-14)
- *2013-12-12* Skeleton King removed shortly before, Legion Commander added, Wraith King added shortly after (Explanatory table for 2013-12-12)
- *2014-01-29* Terrorblade, Phoenix released (Explanatory table for 2014-01-19)
- *2014-02-05* Year Beast Brawl (special game mode) (Explanatory table for 2014-02-05)
- *2014-11-20* Oracle released (Explanatory table for 2014-11-20)
- *2015-02-18* Year Beast Brawl (special game mode) (Explanatory table for 2015-02-18)
- *2015-04-30* Major balance changes (Explanatory table for 2015-04-30)
- *2015-05-03* No major changes, but The Summit 3 Tournament tickets released (Explanatory table for 2015-05-03)
- *2015-09-25* Major balance changes (prev. day) (Explanatory table for 2015-09-25)
- *2015-12-16* Arc Warden released (Explanatory table for 2015-12-16)

sures such as euclidian distance. We did not see significant enough differences to warrant further investigation at this stage.

We produced some additional material such as the pick rate for the most popular heroes (see figure 5) and for the heroes with the highest variability (see figure 6).

Figure 5: Top 10 Heroes by Pick Rate Average (full size)

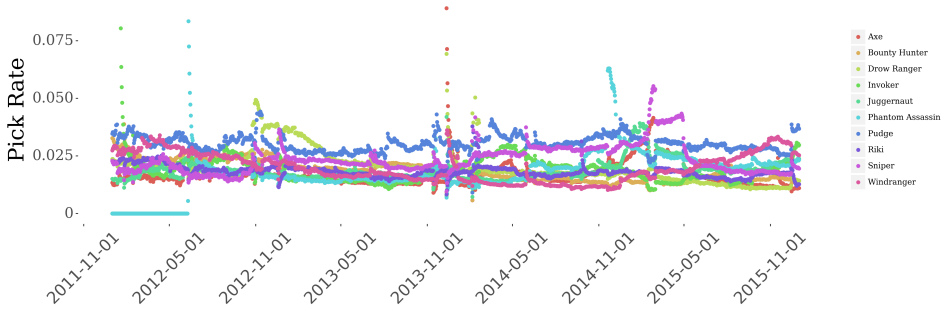
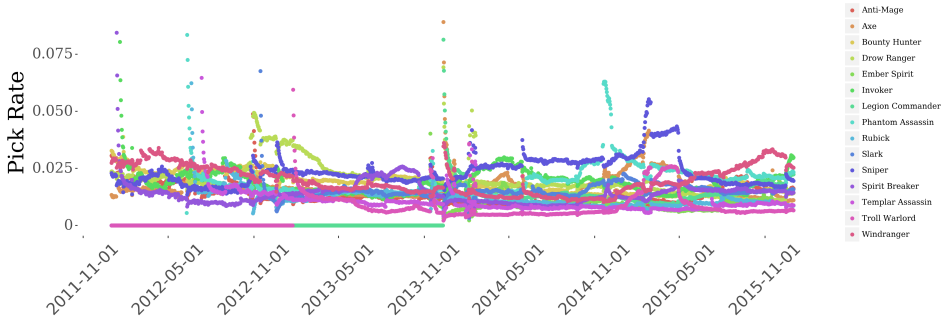


Figure 6: Top 10 Heroes by Variation (Std. Dev.) (full size)



4 Discussion

We have identified a number of shortcomings and good areas for future work.

Game Types. Currently all game modes were treated as one, but this includes game modes where players are playing versus the AI, random heroes modes, and various draft modes where picks and bans are used to choose heroes. This was largely acceptable for our initial results, but

we were advised that this was responsible for some of the most significant spikes which related to the limited time release of unique game modes. As such, while we are correctly detecting major metagame events, it might be beneficial to normalize away such unique game modes and focus only on the major, more traditional, game modes to produce more balance-oriented metrics.

Win Ratios. We have started to look at wins and losses for each heroes. This has the potential to provide us with greater insights on trends in the metagame as it would not only allow us to evaluate how often a hero is chosen, but also how successful that hero is in actual gameplay. This might be a better indicator of underlaying changes in the game rather than changes in popularity. The divergence between these two metrics could provide insight as to the inflexibility of some heroes' popularity. We have produced some preliminary results that indicate that win ratio might be far more nuanced than pick rate. For instance, we can see that examining the win ratios of the top 10 most popular heroes (see figure 7) will yield very similar characteristics to the overall winratio differences chart (see figure 8), whereas doing so for just the pick rate of the top 10 most popular champion does not do so.

Figure 7: Difference for Top 10 Most Popular Heroes, Win Ratio (full size)

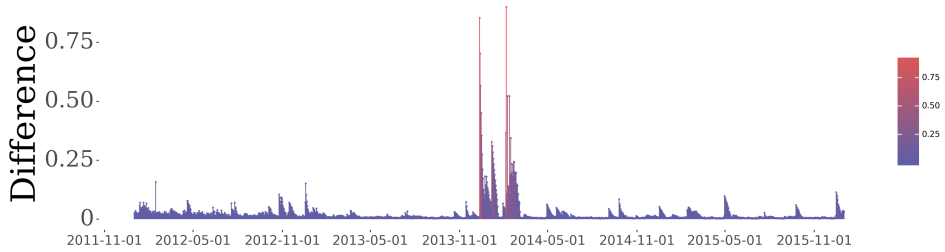
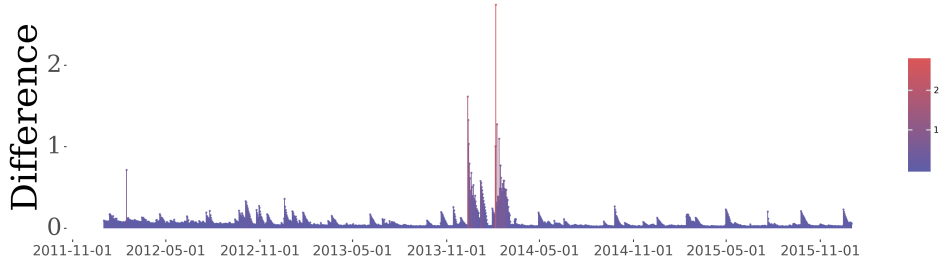


Figure 8: Difference for All Heroes, Win Ratio (full size)



Skill Levels. We suspect players of different skill levels might react with different timeframes or different intensity to metagame changes. For instance, we detected no overly significant changes connected to tournaments but wonder whether players of a higher skill level might react to such subtler signals. We could avail ourselves of the *matches_skill* file and/or infer the skill level from the game mode played to explore these sort of results.

Increased Granularity. We would like to reprocess the data for a hourly-granularity rather than daily to be able to explore reaction times to patches in more detail.

Interactive Visualization. We have found it challenging to explore our static visualization because of their high dimensionality, and suspect that we might be able to produce a much clearer visualization if we used interactive technologies such as D3.js that would allow for selectively choose the data points to visualize.

Roles. As we noted in the Introduction, Dota 2 fill different & specific roles. We would like to investigate whether we could cluster the heroes or perform some correlation analysis to discover the roles from the underlaying data such as heroes that are picked together, heroes that are picked in response to other heroes, and so on. We could compare this data with data provided by subject matter experts as to what they perceive the roles to be, or lists available online from that timeperiod.

5 Three Questions

5.1 What is the Curse of Dimensionality?

"[E]xponential growth in data causes high sparsity in the data set and unnecessarily increases storage space and processing time" (Pore, 2017)

5.2 Can you explain the difference between Granularity reduction and Dimensionality reduction?

Dimensionality reduction trims the number of attributes recorded (=less columns) whereas to reduce granularity is to reduce the level of detail at which the data is recorded (= less rows).

5.3 What does the size of a granule in represent in granular computing?

The (inverse) degree of abstraction of your model. The larger your granules are, the higher the abstraction in your model. (Yao, 2004)

References

Eden, R., Parent, J., Randall, J., Friedman, E. D., & Trove Development Team. (2013, February). *GNU Trove / trove4j.*

Iyer, E. K., Sood, S., Gupta, N., & Panda, T. (2013, December). What drives Big Data Analytics to Cloud. *International Journal of Consumer & Business Analytics*, *2*, 68-84.

Jin, X., Wah, B. W., Cheng, X., & Wang, Y. (2015, June). Significance and Challenges of Big Data Research. *Big Data Research*, *2*(2), 59-64. doi: 10.1016/j.bdr.2015.01.006

Lussetti, M., & Fraser, D. (2019, March). Big Data Reduction: Lessons Learned from Analyzing One Billion Dota 2 Matches. In *14th annual TRU Undergraduate Research & Innovation Conference.* Kamloops, Canada.

Namiot, D. (2015, August). On Big Data Stream Processing. *International Journal of Open Information Technologies*, *3*(8), 48-51.

Popma, R. (2019, February). *Picocli.*

Pore, P. (2017, April). *Must-Know: What is the curse of dimensionality?* https://www.kdnuggets.com/2017/04/must-know-curse-dimensionality.html.

Roger, A. (2019, March). *SimpleFlatMapper.*

Rucker Jones, A., Conway, S., & opencsv Contributors. (2019, February). *Opencsv.*

The OpenDota Project. (2014, August). *FAQ.* https://blog.opendota.com/2014/08/01/faq/.

The OpenDota Project. (2017, March). *Data Dump (March 2011 to March 2016).* https://blog.opendota.com/2017/03/24/datadump2/.

uniVocity Software Pty Ltd. (2018, October). *Comparisons among all Java-based CSV parsers in existence: uniVocity/csv-parsers-comparison.* univocity.

uniVocity Software Pty Ltd. (2019, February). *uniVocity-parsers.* univocity.

ur Rehman, M. H., Liew, C. S., Abbas, A., Jayaraman, P. P., Wah, T. Y., & Khan, S. U. (2016, December). Big Data Reduction Methods: A

Survey. *Data Science and Engineering*, *1*(4), 265-284. doi: 10.1007/s41019-016-0022-0

Vorontsov, M. (2014, July). *Trove library: Using primitive collections for performance.*

Yao, Y. (2004). Granular Computing. In *Proceedings of The 4th Chinese National Conference on Rough Sets and Soft Computing* (Vol. 31, p. 1-5).

Appendixes

Appendix A: matches_condenser -> Main.java

See on GitHub

```
package com.marcolussetti.opendotamatchescondenser;

import com.jsoniter.JsonIterator;
import com.jsoniter.any.Any;
import com.jsoniter.output.JsonStream;
import gnu.trove.map.hash.THashMap;
import gnu.trove.set.hash.THashSet;
import org.simpleflatmapper.csv.CsvParser;
import picocli.CommandLine;
import picocli.CommandLine.Option;
import picocli.CommandLine.Command;
import picocli.CommandLine.Parameters;

import java.io.*;
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.*;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;

@Command(description = "Process OpenDota Matches File",
        name = "processopendota",
        mixinStandardHelpOptions = true,
        version = "processopendota 0.3")
class ProcessOpenDota implements Callable<Void> {
    // ARGUMENTS
    @Option(names = {"-x", "--extract-to-json"},
```

```
28              description = "Extract an existing .ser file to a
                ↪  JSON file.")
29      private File extractToJson = null;
30
31      @Option(names = {"-c", "--condense"},
32              description = "Condense the input openDota CSV
                ↪  file. If file is GunZipped (.gz), extract it
                ↪  first.")
33      private File condense = null;
34
35      @Option(names = {"-o", "--only-count"},
36              description = "Only count picks, do not record wins
                ↪  & losses")
37      private Boolean onlyCount = false;
38
39      @Parameters(paramLabel = "OUTPUT",
40              description = "Output file for either extract or
                ↪  condense")
41      private File output = null;
42
43      // CONSTANTS
44      public static final int MATCHES_NO = 1191768403;
45      public static final int DAYS_NO = 1870;
46      public static final int REPORT_THRESHOLD = 1000000; //
        ↪  Report progress every million rows
47      public static final int SERIALIZE_THRESHOLD = 10000000; //
        ↪  Serialize every 10 million rows
48
49      // VARIABLES
50      // Keep track of progress
51      private LocalDateTime startOfParsing;
52      private THashSet<Long> allDates = new THashSet<>();
53      private int recordCounter = 0;
54      // Store the data {date: Long, {hero#: int -> picks# int}}
55      private THashMap<Long, THashMap<Integer, Integer[]>> data =
        ↪  new THashMap<>();
56
57      private void condenseInputFile(File input, File output,
        ↪  boolean onlyCount) {
58          this.startOfParsing = LocalDateTime.now();
```

```
59
60          // Main loop!
61          FileReader fileReader;
62          try {
63              fileReader = new FileReader(input);
64              Iterator<String[]> csvReader =
                ↪  CsvParser.iterator(fileReader);
65              String[] headers = csvReader.next();
66              // Iterate through stuff
67              while (csvReader.hasNext()) {
68                  String[] row = csvReader.next();
69
70                  parseRow(row, onlyCount);
71
72                  if (recordCounter % REPORT_THRESHOLD == 0) {
73                      reportProgress(this.recordCounter,
                        ↪  this.allDates.size(),
                        ↪  this.startOfParsing);
74
75                      if (recordCounter % SERIALIZE_THRESHOLD ==
                        ↪  0) {
76                          String destFolder = output.getParent();
77                          String[] destFile =
                            ↪  output.getName().split("\\.");
78                          File outputFile = new File(destFolder +
                            ↪  File.separator + destFile[0] + "_"
                            ↪  + (recordCounter /
                            ↪  SERIALIZE_THRESHOLD) + "." +
                            ↪  destFile[1]);
79
80                          serializeData(data, outputFile);
81                      }
82                  }
83              }
84
85              reportProgress(this.recordCounter,
                ↪  this.allDates.size(), this.startOfParsing);
86              serializeData(data, output);
87          } catch (IOException e) {
88              e.printStackTrace();
```

```
89          }
90
91      }
92
93      private void extractToJson(File input, File output) {
94          THashMap<Long, THashMap<Integer, Integer[]>> hashMap =
            ↪  deserializeData(input);
95
96          writeJSON(hashMap, output);
97      }
98
99      private void parseRow(String[] row, boolean onlyCount) {
100         // Extract relevant fields
101         long startTime = Long.parseLong(row[3]);
102         String pgroup = row[26];
103         boolean radiantWin = row[2].equals("t");
104
105         // Parse date
106         Long date = extractDate(startTime).toEpochDay();
107
108         // Parse picks
109         ArrayList<Integer[]> heroesPicked =
            ↪  extractHeroesPicked(pgroup, radiantWin);
110
111         // Update copy of local map
112         THashMap<Integer, Integer[]> todayPicks =
            ↪  this.data.getOrDefault(date, new THashMap<Integer,
            ↪  Integer[]>());
113         heroesPicked.forEach(heroRecord -> {
114             int hero = heroRecord[0];
115             boolean won = heroRecord[1] == 1;
116
117             Integer[] counts = todayPicks.getOrDefault(hero,
                ↪  new Integer[]{0, 0});
118             if (onlyCount || won)
119                 counts[0] += 1;
120             else
121                 counts[1] += 1;
122             todayPicks.put(hero, counts);
123         });
```

```
124
125             // Push to global map
126             this.data.put(date, todayPicks);
127
128             // Tracking progress
129             allDates.add(date);
130             recordCounter++;
131         }
132
133         private static LocalDate extractDate(long
            ↪  epochTimeInSeconds) {
134             return LocalDateTime.ofInstant(
135                     Instant.ofEpochSecond(epochTimeInSeconds),
136                     ZoneId.of("UTC")
137             ).toLocalDate();
138         }
139
140         private static ArrayList<Integer[]>
            ↪  extractHeroesPicked(String jsonInput, boolean
            ↪  radiantWon) {
141             ArrayList<Integer[]> heroes = new ArrayList<>();
142
143             JsonIterator iterator = JsonIterator.parse(jsonInput);
144             Map<String, Any> jsonObject = null;
145             try {
146                 jsonObject = iterator.read(Any.class).asMap();
147             } catch (IOException e) {
148                 e.printStackTrace();
149             }
150
151             jsonObject.forEach((index, object) -> {
152                 int heroId = object.get("hero_id").toInt();
153                 boolean isRadiant =
                    ↪  object.get("player_slot").toInt() <= 127;
154                 Integer[] heroRecord = {heroId, ((isRadiant &&
                    ↪  radiantWon) || (!isRadiant && !radiantWon)) ? 1
                    ↪  : 0 };
155                 heroes.add(heroRecord);
156             });
157
```

```
158         return heroes;
159
160     }
161
162     private static void reportProgress(int recordCounter, int
        ↪  days, LocalDateTime startOfParsing) {
163         Duration elapsed = Duration.between(startOfParsing,
            ↪  LocalDateTime.now());
164         long elapsedMillis = elapsed.toMillis();
165         DateTimeFormatter dtf =
            ↪  DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
166         double rowsPerSec = (double) recordCounter /
            ↪  elapsedMillis * 1000;
167
168         System.out.printf(
169                 "\n%s (%s elapsed - %s remaining) | %9.2f
                    ↪  rows/s | %,6.2f million rows (%6.2f%%)| %4d
                    ↪  days (%6.2f%%)",
170                 dtf.format(LocalDateTime.now()),
                    ↪  // current time
171                 formatTimeDifference(elapsedMillis),
                    ↪  // elapsed time
172                 formatTimeDifference((long) ((MATCHES_NO -
                    ↪  recordCounter) / rowsPerSec * 1000)),//
                    ↪  remaining time (est.)
173                 (double) recordCounter / elapsedMillis * 1000,
                    ↪  // rows per second
174                 (double) recordCounter / 1000000,
                    ↪  // rows processed (mils)
175                 (double) recordCounter / MATCHES_NO * 100,
                    ↪  // % of rows processed
176                 days,
                    ↪  // days tracked
177                 days / (float) DAYS_NO * 100
                    ↪  // % of days tracked
178         );
179     }
180
181     private static String formatTimeDifference(long millis) {
182         // From https://stackoverflow.com/a/44142896/6238740
```

```
183             return String.format(
184                 "%02d:%02d:%02d",
185                 TimeUnit.MILLISECONDS.toHours(millis),
186                 TimeUnit.MILLISECONDS.toMinutes(millis) -
                    ↪  TimeUnit.HOURS.toMinutes(
187                     TimeUnit.MILLISECONDS.toHours(millis)
188                 ),
189                 TimeUnit.MILLISECONDS.toSeconds(millis) -
                    ↪  TimeUnit.MINUTES.toSeconds(
190                     TimeUnit.MILLISECONDS.toMinutes(millis)
191                 )
192             );
193         }
194 
195         private static void serializeData(THashMap<Long,
            ↪  THashMap<Integer, Integer[]>> data, File output) {
196 
197             // From
                ↪  https://beginnersbook.com/2013/12/how-to-serialize-hashmap-in-java/
198             FileOutputStream fos = null;
199             try {
200                 fos = new FileOutputStream(output);
201                 ObjectOutputStream oos = new
                    ↪  ObjectOutputStream(fos);
202                 oos.writeObject(data);
203                 oos.close();
204                 fos.close();
205             } catch (IOException e) {
206                 e.printStackTrace();
207             }
208             System.out.print("\n> Saved data to " +
                ↪  output.getAbsolutePath());
209         }
210 
211         public static THashMap<Long, THashMap<Integer, Integer[]>>
            ↪  deserializeData(File file) {
212             // From
                ↪  https://beginnersbook.com/2013/12/how-to-serialize-hashmap-in-java/
213             THashMap<Long, THashMap<Integer, Integer[]>> hashMap;
214             try {
```

```
215             FileInputStream fis = new FileInputStream(file);
216             ObjectInputStream ois = new ObjectInputStream(fis);
217             hashMap = (THashMap<Long, THashMap<Integer,
                ↪  Integer[]>>) ois.readObject();
218             ois.close();
219             fis.close();
220         } catch (IOException ioe) {
221             ioe.printStackTrace();
222             return null;
223         } catch (ClassNotFoundException c) {
224             System.out.println("Class not found");
225             c.printStackTrace();
226             return null;
227         }
228         return hashMap;
229     }
230
231     public static void writeJSON(THashMap<Long,
        ↪  THashMap<Integer, Integer[]>> hashMap, File outputFile)
        ↪  {
232
233         String output = JsonStream.serialize(hashMap);
234         try {
235             outputFile.createNewFile();
236         } catch (IOException e) {
237             e.printStackTrace();
238         }
239
240         try (PrintStream out = new PrintStream(new
            ↪  FileOutputStream(outputFile))) {
241             out.print(output);
242             out.flush();
243         } catch (FileNotFoundException e) {
244             e.printStackTrace();
245         }
246
247     }
248
249     public static void main(String[] args) {
250         CommandLine.call(new ProcessOpenDota(), args);
```

```
251         }
252
253         @Override
254         public Void call() throws Exception {
255             // BUSINESS LOGIC
256
257             if (extractToJson == null && condense == null) {
258                 System.out.println("Well you need to select
                    ↪  something... try --help");
259                 return null;
260             }
261             if (extractToJson != null && condense != null) {
262                 System.out.println("Can't have it both ways... try
                    ↪  --help");
263                 return null;
264             }
265             if (output == null) {
266                 System.out.println("Must provide an output file!");
267                 return null;
268             }
269
270             if (extractToJson != null) {
271                 System.out.println("Converting from SER to JSON");
272                 extractToJson(extractToJson, output);
273                 System.out.println("Conversion complete: " +
                    ↪  output.getAbsolutePath());
274             }
275             if (condense != null) {
276                 System.out.println("Condensing from CSV to SER");
277                 condenseInputFile(condense, output, onlyCount);
278                 System.out.println("Condensing complete: " +
                    ↪  output.getAbsolutePath());
279             }
280
281             return null;
282         }
283     }
```

Appendix B: json_to_csv -> opendota_jsontocsv.py

See on GitHub

```
#!/usr/bin/env python3
"""OPENDOTA_JSONTOCSV

Usage:
    opendota_jsontocsv.py JSON_INPUT_FILE CSV_OUTPUT_FILE [--heroes=(names|numbers)] [(-n | --normalize)] [(-r | --remove-low-counts)]
    opendota_jsontocsv.py JSON_INPUT_FILE CSV_OUTPUT_FILE --picks-by-hero [--remove-low-counts] [--heroes=(names|numbers)] [(-r | --remove-low-counts)]
    opendota_jsontocsv.py JSON_INPUT_FILE CSV_OUTPUT_FILE --picks-by-date [--remove-low-counts] [(-r | --remove-low-counts)]
    opendota_jsontocsv.py (-h | --help)
    opendota_jsontocsv.py --version

Options:
    -h --help                   Show this screen.
    --version                   Show version.
    --heroes=(names|numbers)    Record heroes by name or number [default: names].
    -n --normalize              Normalize picks as proportion of picks per day.
    -r --remove-low-counts          Removes early records (pre 2011-11-22) as they have lower volumes of recorded matches.
    --picks-by-date             Export the number of picks for each day to CSV.
    --picks-by-hero             Export the number of picks for each hero to CSV.
"""
import datetime

import requests
import pandas as pd
from docopt import docopt

```

```
26
27  if __name__ == '__main__':
28      arguments = docopt(__doc__, version='OpenDotaDumpTools
        ↪  JsonToCsv 0.2')
29
30      print("Starting OpenDotaDumpTools...")
31
32      df = pd.read_json(arguments["JSON_INPUT_FILE"])
33      print("JSON input loaded")
34
35      # Clean up data
36      df = df.transpose()  # Rotate so rows = time
37      df = df.fillna(0)  # Replace missing values with 0
38      df = df.drop(0, 0)  # Remove entries with missing date
        ↪  (1970)
39      df = df.drop(0, 1)  # Remove entries with a missing hero
        ↪  (0)
40      df.index = [datetime.datetime(1970, 1, 1, 0, 0) +
        ↪  datetime.timedelta(index - 1)
41                  for index in df.index]  # Convert index (epoch
                    ↪  days) to time
42      df = df.reindex(sorted(df.columns), axis=1)  # Order
        ↪  columns by hero #, ascending
43      df = df.sort_index(axis=0)  # Order rows by date, ascending
44      for column in df.columns:  # Convert all values from float
        ↪  to integer
45          df[column] = df[column].astype('int64')
46      print("Input cleaned")
47
48      if arguments["--remove-low-counts"]:
49          df = df.loc[df.index > '2011-11-22 00:00:00']
50          print("Data for days previous to 2011-11-23 removed")
51
52      # EXPORT
53      if arguments["--picks-by-date"]:
54          picks_by_day = df.sum(axis=1)
55          picks_by_day.to_csv(arguments["CSV_OUTPUT_FILE"])
56          print("Exported picks by date data to
            ↪  {}".format(arguments["CSV_OUTPUT_FILE"]))
57          exit(0)  # DONE!
```

```
58
59      if arguments["--heroes"] == "names":
60          # Fetch heroes from OpenDota API
61          heroes_json =
            ↪  requests.get("http://api.opendota.com/api/heroes/").json()
62          heroes = {hero["id"]: hero for hero in heroes_json}
63          df.columns = [heroes[column]["localized_name"] for
            ↪  column in df.columns]
64          print("Heroes ids replaced with heroes names")
65
66      # EXPORT
67      if arguments["--picks-by-hero"]:
68          picks_by_day = df.sum(axis=0)
69          picks_by_day.to_csv(arguments["CSV_OUTPUT_FILE"])
70          print("Exported picks by hero data to
            ↪  {}".format(arguments["CSV_OUTPUT_FILE"]))
71          exit(0)  # DONE!
72
73      if arguments["--normalize"]:
74          df = df.div(df.sum(axis=1), axis=0)
75          print("Values normalized")
76
77      # EXPORT
78      df.to_csv(arguments["CSV_OUTPUT_FILE"])
79      print("Exported data to
        ↪  {}".format(arguments["CSV_OUTPUT_FILE"]))
```

Appendix C: Explanatory Tables

Figure 9: Explanatory table for 2012-01-12

2012-01-12			Position #	Day's difference
			13	0.32048
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Spirit Breaker	0.26331	0.08439	0.08439	0.00000
Silencer	0.20795	0.06665	0.06665	0.00000
Axe	0.02655	0.00851	0.02138	0.01287
Anti-Mage	0.02358	0.00756	0.02281	0.03036
Earthshaker	0.01796	0.00576	0.01530	0.02106
Pudge	0.01612	0.00517	0.03069	0.03585
Tidehunter	0.01538	0.00493	0.01609	0.02102
Night Stalker	0.01519	0.00487	0.01533	0.02020
Huskar	0.01404	0.00450	0.01863	0.02313
Nature's Prophet	0.01400	0.00449	0.02038	0.02487

Figure 10: Explanatory table for 2012-06-11

2012-06-11			Position #	Day's difference
			9	0.39309
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Phantom Assassin	0.21126	0.08304	0.08343	0.00039
Chaos Knight	0.15815	0.06217	0.06254	0.00037
Gyrocopter	0.12698	0.04991	0.05026	0.00034
Anti-Mage	0.02068	0.00813	0.01151	0.01964
Drow Ranger	0.01755	0.00690	0.01586	0.02276
Bounty Hunter	0.01620	0.00637	0.01950	0.02587
Ogre Magi	0.01619	0.00636	0.01440	0.02076
Pudge	0.01408	0.00553	0.02706	0.03259
Riki	0.01301	0.00511	0.01443	0.01955
Lifestealer	0.01288	0.00506	0.01022	0.01529

Figure 11: Explanatory table for 2012-07-26

2012-07-26			Position #	Day's difference
			21	0.26003
Hero	**Portion of day's shift**	**Manhattan distance**	**Pick for day**	**14 days average**
Nyx Assassin	0.22387	0.05821	0.05821	0.00000
Keeper of the Light	0.17856	0.04643	0.04643	0.00000
Visage	0.09718	0.02527	0.02527	0.00000
Templar Assassin	0.05146	0.01338	0.01818	0.03156
Nature's Prophet	0.01200	0.00312	0.01797	0.02109
Luna	0.01082	0.00281	0.01132	0.01413
Disruptor	0.01016	0.00264	0.00603	0.00867
Phantom Assassin	0.01014	0.00264	0.01861	0.02125
Venomancer	0.00908	0.00236	0.01055	0.01291
Windranger	0.00905	0.00235	0.02327	0.02562

Figure 12: Explanatory table for 2012-10-30

2012-10-30			Position #	Day's difference
			5	0.43497
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Anti-Mage	0.08130	0.03536	0.04881	0.01345
Phantom Lancer	0.05785	0.02516	0.03701	0.01185
Drow Ranger	0.03940	0.01714	0.04779	0.03065
Storm Spirit	0.03200	0.01392	0.02299	0.00907
Centaur Warrunner	0.02757	0.01199	0.02739	0.01540
Tinker	0.02658	0.01156	0.01800	0.00644
Ursa	0.02631	0.01144	0.02338	0.01193
Phantom Assassin	0.02540	0.01105	0.02810	0.01705
Queen of Pain	0.02537	0.01104	0.02342	0.01238
Clinkz	0.02454	0.01068	0.02272	0.01204

Figure 13: Explanatory table for 2012-12-19

2012-12-19			Position #	Day's difference
			14	0.29522
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Ursa	0.06175	0.01823	0.03093	0.01271
Bounty Hunter	0.04558	0.01346	0.01195	0.02541
Magnus	0.03464	0.01023	0.02035	0.01012
Undying	0.03297	0.00973	0.01813	0.00839
Jakiro	0.03140	0.00927	0.02182	0.01255
Riki	0.02853	0.00842	0.01175	0.02017
Wraith King	0.02689	0.00794	0.02036	0.01242
Death Prophet	0.02441	0.00721	0.01931	0.01210
Windranger	0.02348	0.00693	0.01481	0.02174
Drow Ranger	0.02033	0.00600	0.03225	0.03825

Figure 14: Explanatory table for 2013-11-14

2013-11-14			Position #	Day's difference
			4	0.51397
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Ember Spirit	0.07823	0.04021	0.04021	0.00000
Earth Spirit	0.05589	0.02872	0.02872	0.00000
Storm Spirit	0.04806	0.02470	0.03307	0.00837
Troll Warlord	0.03499	0.01798	0.02451	0.00653
Anti-Mage	0.03315	0.01704	0.02937	0.01233
Shadow Shaman	0.02605	0.01339	0.02134	0.00795
Lion	0.02463	0.01266	0.02584	0.01318
Tinker	0.02234	0.01148	0.01670	0.00522
Queen of Pain	0.01931	0.00992	0.01984	0.00992
Bounty Hunter	0.01805	0.00928	0.01138	0.02066

Figure 15: Explanatory table for 2013-12-12

2013-12-12			Position #	Day's difference
			1	0.97501
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Legion Commander	0.08344	0.08135	0.08135	0.00000
Axe	0.07719	0.07526	0.08908	0.01382
Drow Ranger	0.05203	0.05073	0.06927	0.01854
Sven	0.05004	0.04879	0.06001	0.01122
Omniknight	0.03848	0.03752	0.04452	0.00700
Shadow Fiend	0.03276	0.03194	0.04239	0.01045
Venomancer	0.02934	0.02861	0.04094	0.01233
Juggernaut	0.02560	0.02496	0.04193	0.01696
Magnus	0.02359	0.02300	0.02886	0.00586
Windranger	0.01847	0.01801	0.03465	0.01664

Figure 16: Explanatory table for 2014-01-19

2014-01-19			Position #	Day's difference
			767	0.03547
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Invoker	0.08742	0.00310	0.02250	0.01940
Pudge	0.05027	0.00178	0.02802	0.02980
Timbersaw	0.04328	0.00154	0.01282	0.01128
Luna	0.04286	0.00152	0.01375	0.01222
Windranger	0.04137	0.00147	0.01401	0.01548
Axe	0.04133	0.00147	0.01213	0.01360
Nyx Assassin	0.03327	0.00118	0.00961	0.00843
Pugna	0.03252	0.00115	0.00739	0.00623
Alchemist	0.03043	0.00108	0.01598	0.01490
Dazzle	0.02707	0.00096	0.00766	0.00670

Figure 17: Explanatory table for 2014-02-05

2014-02-05			Position #	Day's difference
			2	0.82423
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Warlock	0.07778	0.06410	0.07286	0.00876
Lich	0.07637	0.06294	0.07287	0.00993
Death Prophet	0.05124	0.04224	0.05481	0.01257
Troll Warlord	0.03721	0.03067	0.03580	0.00513
Venomancer	0.03419	0.02818	0.03914	0.01096
Templar Assassin	0.03315	0.02732	0.03548	0.00816
Shadow Shaman	0.02931	0.02416	0.03355	0.00940
Drow Ranger	0.02897	0.02387	0.04348	0.01961
Visage	0.02575	0.02123	0.02479	0.00356
Slardar	0.02462	0.02029	0.02953	0.00924

Figure 18: Explanatory table for 2014-11-20

2014-11-20			Position #	Day's difference
			50	0.18819
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Oracle	0.26931	0.05068	0.05078	0.00010
Phantom Assassin	0.17439	0.03282	0.06201	0.02919
Sniper	0.01496	0.00282	0.02632	0.02914
Anti-Mage	0.01163	0.00219	0.00958	0.01177
Drow Ranger	0.01143	0.00215	0.01506	0.01721
Invoker	0.01052	0.00198	0.01781	0.01979
Nature's Prophet	0.01030	0.00194	0.00968	0.01162
Medusa	0.00947	0.00178	0.00525	0.00703
Mirana	0.00938	0.00177	0.01630	0.01806
Spectre	0.00904	0.00170	0.00736	0.00906

Figure 19: Explanatory table for 2015-02-18

2015-02-18			Position #	Day's difference
			26	0.24633
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Warlock	0.07194	0.01772	0.02646	0.00874
Zeus	0.07137	0.01758	0.03398	0.01640
Undying	0.06689	0.01648	0.02448	0.00800
Omniknight	0.05957	0.01467	0.02664	0.01196
Sniper	0.04918	0.01212	0.04793	0.03581
Juggernaut	0.03927	0.00967	0.02425	0.03392
Axe	0.02907	0.00716	0.03890	0.03174
Troll Warlord	0.02702	0.00666	0.02365	0.01700
Death Prophet	0.02391	0.00589	0.01403	0.00814
Pudge	0.02320	0.00572	0.02218	0.02789

Figure 20: Explanatory table for 2015-04-30

2015-04-30			Position #	Day's difference
			74	0.15652
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Alchemist	0.09490	0.01485	0.02022	0.00537
Troll Warlord	0.06202	0.00971	0.01547	0.02518
Sniper	0.06053	0.00947	0.03270	0.04217
Nature's Prophet	0.03072	0.00481	0.01249	0.00768
Legion Commander	0.03031	0.00474	0.01436	0.00961
Juggernaut	0.02825	0.00442	0.02142	0.02584
Storm Spirit	0.02575	0.00403	0.01711	0.02115
Wraith King	0.02404	0.00376	0.01458	0.01082
Lifestealer	0.02376	0.00372	0.01155	0.00784
Axe	0.02244	0.00351	0.02152	0.02503

Figure 21: Explanatory table for 2015-05-03

2015-05-03			Position #	Day's difference
			150	0.11656
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Sniper	0.09893	0.01153	0.02816	0.03969
Troll Warlord	0.08760	0.01021	0.01247	0.02268
Undying	0.05405	0.00630	0.01297	0.00667
Alchemist	0.05239	0.00611	0.01413	0.00802
Juggernaut	0.04088	0.00477	0.01988	0.02465
Storm Spirit	0.03525	0.00411	0.01593	0.02004
Axe	0.03475	0.00405	0.02004	0.02409
Ursa	0.02274	0.00265	0.01224	0.00959
Luna	0.02244	0.00262	0.00999	0.00737
Night Stalker	0.02046	0.00239	0.00846	0.00607

Figure 22: Explanatory table for 2015-09-25

2015-09-25			Position #	Day's difference
			203	0.09711
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Storm Spirit	0.08104	0.00787	0.01680	0.02467
Bloodseeker	0.07088	0.00688	0.01978	0.02666
Leshrac	0.06080	0.00590	0.00645	0.01235
Lina	0.04945	0.00480	0.01971	0.02451
Invoker	0.03631	0.00353	0.01733	0.01380
Bounty Hunter	0.03535	0.00343	0.01498	0.01841
Phantom Assassin	0.03409	0.00331	0.02444	0.02113
Ogre Magi	0.03172	0.00308	0.01064	0.00756
Lifestealer	0.02933	0.00285	0.01062	0.00777
Spectre	0.02710	0.00263	0.01051	0.00787

Figure 23: Explanatory table for 2015-12-16

2015-12-16			Position #	Day's difference
			23	0.2559
Hero	**% of total distance**	**Distance (Manhattan)**	**Pick ratio**	**14 days average**
Arc Warden	0.17289	0.04425	0.04458	0.00033
Zeus	0.06070	0.01553	0.02745	0.01192
Pudge	0.05079	0.01300	0.03865	0.02565
Riki	0.04504	0.01153	0.02934	0.01782
Windranger	0.02524	0.00646	0.02421	0.03067
Sniper	0.02140	0.00548	0.02251	0.01703
Drow Ranger	0.02107	0.00539	0.01659	0.01120
Tusk	0.01822	0.00466	0.01175	0.01641
Ember Spirit	0.01802	0.00461	0.01291	0.01752
Bristleback	0.01773	0.00454	0.01569	0.01115

Appendix D: analysis -> LookupSpikes.ipynb

See on GitHub - Run in Google Colab

Processing

Imports & Configuration

```
# Update pandas, just in case
!pip install pandas -U

# If plotnine is not installed:
!pip install plotnine

# If using on google colab, might need to update statsmodels version
!pip install statsmodels -U

# If not installed
!pip install requests
```

Constants for configuration

```
CSV_INPUT_FILE = ("https://raw.githubusercontent.com/marcolussetti"
"/processopendota/master/data/heroes_picks_csvs/"
"stable-picks_heroes-names_normalized.csv")
OPENDOTA_API_HEROES_ENDPOINT = "https://api.opendota.com/api/heroes/"

import pandas as pd
import requests
from plotnine import *
from scipy.spatial import distance
from datetime import datetime, timedelta

%matplotlib inline
```

Import Data

```
# Load input csv
df = pd.read_csv(CSV_INPUT_FILE, index_col=0)
```

Examine the data - Overall heroes metrics

```
# Most popular heroes overall (mean)
heroes_most_popular =
↪  df.mean().sort_values(ascending=False)[:10]  # Average of
↪  normalized pick frequency

# Heroes with the most variation
heroes_most_variation =
↪  df.std().sort_values(ascending=False)[:15]  # Standard
↪  deviation of pick frequency

# Heroes with the least variation
heroes_least_variation =
↪  df.std().sort_values(ascending=False)[:10]  # Standard
↪  deviation of pick frequency
```

Examine the data - Reformat the data for easy graphing

```
df_expl_graph = df.copy(deep=True)
# Condense values
df_expl_graph = df_expl_graph.stack()
df_expl_graph = df_expl_graph.reset_index()

df_expl_graph.columns = ["Day", "Hero", "Frequency"]

df_expl_graph["Day"] =
↪  df_expl_graph["Day"].apply(pd.to_datetime)

df_expl_graph["Week"] = df_expl_graph["Day"].apply(lambda date:
↪  "{}-{}".format(date.year,date.week))
df_expl_graph["Month"] = df_expl_graph["Day"].apply(lambda
↪  date: "{}-{}".format(date.year,date.month))
df_expl_graph["Year"] = df_expl_graph["Day"].apply(lambda date:
↪  date.year)

df_most_popular_graph =
↪  df_expl_graph[df_expl_graph["Hero"].isin(heroes_most_popular.keys())]
df_most_variation_graph =
↪  df_expl_graph[df_expl_graph["Hero"].isin(heroes_most_variation.keys())]
```

Examine the data - Graphs

```
all_day_plot = (ggplot(df_expl_graph, aes(x="Day",
↪  y="Frequency", color="Hero", group=1))
              +geom_point()
             )
# all_day_plot
# all_day_plot.save("all_day_plot.png", width=40, height=32,
↪  dpi=300, limitsize=False)

all_day_stacked_plot = (ggplot(df_expl_graph, aes(x="Day",
↪  y="Frequency"))
              +geom_area(aes(fill="Hero"))
             )
# all_day_stacked_plot
# all_day_stacked_plot.save("all_day_stacked_plot.png",
↪  width=44, height=12, dpi=300, limitsize=False)
```

Examine the data - Detect changes over time

```
def previous_distribution_vector(df, date_start, date_end,
↪  average_function="mean"):
  df_filtered = df[(df["Day"] >= date_start) & (df["Day"] <
  ↪  date_end)]

  if average_function == "median":
    result =
    ↪  df_filtered.groupby(["Hero"]).median()[["Frequency"]]
  else:
    result =
    ↪  df_filtered.groupby(["Hero"]).mean()[["Frequency"]]

  return result.to_dict()["Frequency"]

def compute_distance(day, previous_period_average,
↪  distance_function=distance.euclidean, weighted=False):
  previous = [value for key, value
              in sorted(previous_period_average.items(),
               ↪  key=lambda x: x[0])]
  current = [value for key, value
              in sorted(day.items(), key=lambda x: x[0])]
```

```
    assert len(previous) == len(current), "Incorrect length:
    ↪  previous-> {}, current-> {}".format(len(previous),
    ↪  len(current))
    if weighted:
      return distance_function(previous, current, previous)
    else:
      return distance_function(previous, current)

def day_difference(df, day,
↪  distance_function=distance.euclidean, length=14,
↪  average_function="mean", weighted=False):
    # Extract vector for day
    day_picks = {record["Hero"]: record["Frequency"] for record
                 in df[df["Day"] == day][["Hero",
                  ↪  "Frequency"]].to_dict('records')
                }
    previous_picks = previous_distribution_vector(df,
    ↪  datetime.strptime(day, '%Y-%m-%d') -
    ↪  timedelta(days=length), day, average_function)

    return compute_distance(day_picks, previous_picks,
    ↪  distance_function, weighted)

def all_days_difference(df,
↪  distance_function=distance.euclidean, length=14,
↪  average_function="mean", weighted=False):
    all_days = [str(d) for d in sorted(set(date.date() for key,
    ↪  date in df["Day"].to_dict().items()))[1:]]

    return {day: day_difference(
        df, day, distance_function=distance_function,
        length=length, average_function="mean")
     for day in all_days}
```

Graph differences

```
# Try to graph differences for top 10 champions by popularity
pop_differences_by_day =
↪  all_days_difference(df_most_popular_graph).items()
sorted_pop_differences_by_day = sorted(pop_differences_by_day,
↪  key=lambda x: x[1], reverse=True)
```

```
df_pop_differences_by_day = pd.DataFrame(pop_differences_by_day)
df_pop_differences_by_day.columns = ["Day", "Difference"]

popular_heroes_differences_plot = (
    ggplot(df_pop_differences_by_day, aes(x="Day", y="Difference", color="Difference"))
    +geom_point()
    +geom_area(aes(fill="Difference"))
)

# popular_heroes_differences_plot
# popular_heroes_differences_plot.save("pop_differences_plot.png", width=44, height=5, dpi=300, limitsize=False)

# Try to graph differences for all heroes

all_differences_by_day = all_days_difference(df_expl_graph).items()
sorted_all_differences_by_day = sorted(all_differences_by_day, key=lambda x: x[1], reverse=True)
df_all_differences_by_day = pd.DataFrame(all_differences_by_day)
df_all_differences_by_day.columns = ["Day", "Difference"]
df_all_differences_by_day.head()

all_heroes_differences_plot = (
    ggplot(df_all_differences_by_day, aes(x="Day", y="Difference", color="Difference"))
    +geom_point()
    +geom_area(aes(fill="Difference"))
)

# all_heroes_differences_plot
# all_heroes_differences_plot.save("all_differences_plot.png", width=44, height=5, dpi=300, limitsize=False)

# What if we weight it?
# Try to graph differences for all heroes
```

```
35
36 all_differences_by_day_weighted =
   ↪ all_days_difference(df_expl_graph, weighted=True).items()
37 sorted_all_differences_by_day_weighted =
   ↪ sorted(all_differences_by_day_weighted, key=lambda x: x[1],
   ↪ reverse=True)
38 df_all_differences_by_day_weighted =
   ↪ pd.DataFrame(all_differences_by_day_weighted)
39 df_all_differences_by_day_weighted.columns = ["Day",
   ↪ "Difference"]
40 df_all_differences_by_day_weighted.head()
41
42 all_heroes_differences_weighted_plot = (
43     ggplot(df_all_differences_by_day_weighted, aes(x="Day",
       ↪ y="Difference", color="Difference"))
44     +geom_point()
45     +geom_area(aes(fill="Difference"))
46 )
47
48 # all_heroes_differences_weighted_plot
49 #
   ↪ all_heroes_differences_plot.save("all_differences_weighted_plot.png",
   ↪ width=44, height=5, dpi=300, limitsize=False)
50
51 # Try to graph differences for all heroes, 28 days
52
53 all_differences_by_day_28 = all_days_difference(df_expl_graph,
   ↪ length=28).items()
54 sorted_all_differences_by_day_28 =
   ↪ sorted(all_differences_by_day_28, key=lambda x: x[1],
   ↪ reverse=True)
55 df_all_differences_by_day_28 =
   ↪ pd.DataFrame(all_differences_by_day_28)
56 df_all_differences_by_day_28.columns = ["Day", "Difference"]
57 df_all_differences_by_day_28.head()
58
59 all_differences_by_day_28_plot = (
60     ggplot(df_all_differences_by_day_28, aes(x="Day",
       ↪ y="Difference", color="Difference"))
61     +geom_point()
```

```
    +geom_area(aes(fill="Difference"))
)

# all_differences_by_day_28_plot
#
↪  all_differences_by_day_28_plot.save("all_differences_28_plot.png",
↪  width=44, height=5, dpi=300, limitsize=False)

# Try to graph differences for all heroes, manhattan distance

all_differences_by_day_manhattan =
↪  all_days_difference(df_expl_graph,
↪  distance_function=distance.cityblock).items()
sorted_all_differences_by_day_manhattan =
↪  sorted(all_differences_by_day_manhattan, key=lambda x:
↪  x[1], reverse=True)
df_all_differences_by_day_manhattan =
↪  pd.DataFrame(all_differences_by_day_manhattan)
df_all_differences_by_day_manhattan.columns = ["Day",
↪  "Difference"]

all_heroes_differences_manhattan_plot = (
    ggplot(df_all_differences_by_day_manhattan, aes(x="Day",
    ↪  y="Difference", color="Difference"))
    +geom_point()
    +geom_area(aes(fill="Difference"))
)

# all_heroes_differences_manhattan_plot
#
↪  all_heroes_differences_manhattan_plot.save("all_differences_manhattan_plot.png",
↪  width=44, height=5, dpi=300, limitsize=False)
```

Poster Graphs

Pick rates by hero (stacked) graph

```
poster_stacked = (
    ggplot(df_expl_graph, aes(x="Day", y="Frequency"))
    +geom_area(aes(fill="Hero"), color="white")
```

```
    +guides(fill=guide_legend(ncol=3, title="Heroes"))
    +scale_x_datetime(date_breaks="6 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    +ggtitle("Heroes Pick Rates (2011-11-22 - 2016-04-23)")
    +xlab("")
    +ylab("Pick Rate")
    +theme(
        text=element_text(family=['serif']),
        axis_text=element_text(size=24.0),
        #axis_text_x=element_text(ha="right"),
        axis_title_y=element_text(size=36.0),
        axis_title_x=element_text(size=0.0),
        legend_title=element_blank(),
        plot_title=element_text(size=72.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu']),#, angle=45),
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
    )
)
# poster_stacked
poster_stacked.save("poster_stacked_white_46x12.png", width=46,
↪  height=12, dpi=300, limitsize=False)
```

Differences graph

```
poster_differences_plot = (
    ggplot(df_all_differences_by_day_manhattan, aes(x="Day",
    ↪  y="Difference", color="Difference"))
    +geom_point()
    +geom_area(aes(fill="Difference"))
    +scale_color_gradient(low="#5D5DA9", high="#DC5657")
    +scale_fill_gradient(low="#5D5DA9", high="#DC5657")
    +scale_x_datetime(date_breaks="6 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    +xlab("")
    +theme(
        text=element_text(family=['serif']),
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
        panel_grid=element_blank(),
```

```
        axis_text=element_text(size=24.0),
        axis_title_y=element_text(size=36.0),
        axis_title_x=element_text(size=0.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu']),
        legend_text=element_text(family=['Dejavu Sans',
        ↪  'Dejavu']),
        legend_title=element_blank(),
    )
)

# poster_differences_plot
poster_differences_plot.save("poster_differences_white_41x5.png",
↪  width=41, height=5, dpi=300, limitsize=False)
```

Differences graph (3mo)

```
poster_differences_plot_3mo = (
    ggplot(df_all_differences_by_day_manhattan, aes(x="Day",
    ↪  y="Difference", color="Difference"))
    +geom_point()
    +geom_area(aes(fill="Difference"))
    +scale_color_gradient(low="#5D5DA9", high="#DC5657")
    +scale_fill_gradient(low="#5D5DA9", high="#DC5657")
    +scale_x_datetime(date_breaks="3 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    +xlab("")
    +theme(
        text=element_text(family=['serif']),
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
        panel_grid=element_blank(),
        axis_text=element_text(size=24.0),
        axis_title_y=element_text(size=36.0),
        axis_title_x=element_text(size=0.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu']),
        legend_text=element_text(family=['Dejavu Sans',
        ↪  'Dejavu']),
        legend_title=element_blank(),
    )
```

```
)

poster_differences_plot_3mo.save("poster_differences_white_41x5_3mo.png", width=41, height=5, dpi=300, limitsize=False)
```

Explore differences

```
# Try to graph differences for all heroes, manhattan distance

all_differences_by_day_manhattan = all_days_difference(df_expl_graph, distance_function=distance.cityblock).items()
differences_by_day_manhattan = all_days_difference(df_expl_graph, distance_function=distance.cityblock)
sorted_all_differences_by_day_manhattan = sorted(all_differences_by_day_manhattan, key=lambda x: x[1], reverse=True)
df_all_differences_by_day_manhattan = pd.DataFrame(all_differences_by_day_manhattan)
df_all_differences_by_day_manhattan.columns = ["Day", "Difference"]

top_differences = {record[0]: {"order": i + 1, "value": record[1]} for i, record in list(enumerate(sorted_all_differences_by_day_manhattan))}

all_differences_by_day_manhattan

top_differences["2015-04-30"]

top_differences
```

Printout Graphs

Most Popular

```
poster_stacked = (
    ggplot(df_most_popular_graph, aes(x="Day", y="Frequency"))
    +geom_area(aes(fill="Hero"), color="white")
    +guides(fill=guide_legend(ncol=1, title="Heroes"))
```

```
    +scale_x_datetime(date_breaks="6 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    +ggtitle("Top 10 Overall Most Popular Heroes")
    +xlab("")
    +ylab("Pick Rate")
    +theme(
        text=element_text(family=['serif']),
        axis_text=element_text(size=16.0),
        #axis_text_x=element_text(ha="right"),
        axis_title_y=element_text(size=24.0),
        axis_title_x=element_text(size=0.0),
        legend_title=element_blank(),
        plot_title=element_text(size=24.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu'], angle=45),
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
    )
)

poster_stacked

# poster_stacked
poster_stacked.save("printout_most_popular_cumulative_15x4_5.png",
↪  width=15, height=4.5, dpi=300, limitsize=False)



poster_stacked = (
    ggplot(df_most_popular_graph, aes(x="Day", y="Frequency"))
    +geom_point(aes(color="Hero"))
    +guides(color=guide_legend(ncol=1, title="Heroes"))
    +scale_x_datetime(date_breaks="6 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    #+ggtitle("Most PopulHeroes Pick Rates (2011-11-22 -
    ↪  2016-04-23)")
    +xlab("")
    +ylab("Pick Rate")
    +theme(
        text=element_text(family=['serif']),
```

```
        axis_text=element_text(size=16.0),
        #axis_text_x=element_text(ha="right"),
        axis_title_y=element_text(size=24.0),
        axis_title_x=element_text(size=0.0),
        legend_title=element_blank(),
        plot_title=element_text(size=24.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu'], angle=45),
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
    )
)

poster_stacked

# poster_stacked
poster_stacked.save("printout_most_popular_point_15x4_5.png",
↪  width=15, height=4.5, dpi=300, limitsize=False)
```

Most variation

```
poster_stacked = (
    ggplot(df_most_variation_graph, aes(x="Day",
    ↪  y="Frequency"))
    +geom_line(aes(color="Hero"))
    +guides(color=guide_legend(ncol=1, title="Heroes"))
    +scale_x_datetime(date_breaks="6 months", minor_breaks=4,
    ↪  limits=["2012-01-01", "2016-01-01"])
    +ggtitle("Heroes With Highest Variability (Std. Dev.)")
    +xlab("")
    +ylab("Pick Rate")
    +theme(
        text=element_text(family=['serif']),
        axis_text=element_text(size=16.0),
        #axis_text_x=element_text(ha="right"),
        axis_title_y=element_text(size=24.0),
        axis_title_x=element_text(size=0.0),
        legend_title=element_blank(),
        plot_title=element_text(size=24.0),
        axis_text_x=element_text(size=18.0, family=['Dejavu
        ↪  Sans', 'Dejavu'], angle=45),
```

```
        panel_background=element_rect(fill="white",
        ↪  colour="white"),
    )
)

poster_stacked


# poster_stacked
poster_stacked.save("printout_most_variation_line_15x4_5.png",
↪  width=15, height=4.5, dpi=300, limitsize=False)
```

Printout Tables

```
KEY_DATES = [
    "2012-01-12", "2012-06-11", "2012-07-26",
    "2012-10-30", "2012-12-19", "2013-11-14",
    "2013-12-12", "2014-01-19", "2014-02-05",
    "2014-11-20", "2015-02-18", "2015-04-30",
    "2015-05-03", "2015-09-25", "2015-12-16"
]

heroes = [hero for hero in df_expl_graph["Hero"]]

two_week_averages = {date:
↪  previous_distribution_vector(df_expl_graph,
↪  datetime.strptime(date, '%Y-%m-%d') - timedelta(days=14),
↪  date, average_function="mean") for date in KEY_DATES}

day_average = {date: {row["Hero"]: row["Frequency"] for id, row
↪  in df_expl_graph[df_expl_graph["Day"] == date][["Hero",
↪  "Frequency"]].iterrows()} for date in KEY_DATES}

day_difference = {date: {hero: abs(frequency -
↪  two_week_averages[date][hero]) for hero, frequency in
↪  day_average[date].items()} for date in KEY_DATES}

```

```
day_difference_top_10 = {date:
    (sorted(list(differences.items()), key=lambda x: x[1],
    reverse=True)[:10]) for date, differences in
    day_difference.items()}

def process_differences(date, differences):
  results = [("Hero", "Portion of day's shift", "Manhattan
    distance",
              "Pick for day", "Average pick for 14 prev day")]

  total_day_difference =
    dict(all_differences_by_day_manhattan)[date]
#   for hero in heroes:
#     if hero in [hero in differences] or hero in
    two_week_averages.keys():
#       results.append((
#           hero,
#           differences[hero] / total_day_difference,
#           differences[hero],
#           day_average[date][hero],
#           two_week_averages[date][hero]
#       ))

  for hero, difference in differences:
    results.append((
        hero,
        difference / total_day_difference,
        difference,
        day_average[date][hero],
        two_week_averages[date][hero]
    ))

  return results

top_10_differences_by_day = {day: process_differences(day,
    differences) for day, differences in
    day_difference_top_10.items()}

top_10_differences_by_day

```

```
# Write to CSV-ish
with open("explain.csv", "w") as f:
  for date, rows in top_10_differences_by_day.items():
    date_row = (date, "Position:
    ↪  {}".format(top_differences[date]["order"]), "Shift:
    ↪  {}".format(dict(all_differences_by_day_manhattan)[date]))
    for date_component in date_row:
      f.write(date_component)
      f.write(",")
    f.write("\n")
    for row in rows:
      for item in row:
        f.write(str(item))
        f.write(",")
      f.write("\n")
```