Predicting Toxic Behaviors in Dota 2

Colin Flaherty

1. Introduction

The accompanying Python script will obtain a random group of suitable Dota 2 public matches, and for each match use a google service to classify the messages sent as toxic or non toxic, and restructure the match data from the given JSON format into a pandas array suitable for more study. I use the numpy, requests, json, time, re, and pandas libraries, as well as various sklearn modules. This document is meant to enable the modification of the accompanying script, such that investigation may be extended and continued.

2. Match Collection

match_ids is a list of IDs of suitable DOTA 2 online matches (that is, public and from regions 1 or 2, which speak English). At line 36 we have the following loop:

```
1  while len(match_ids) < 400:
2      time.sleep(1)
3      mr = requests.get( 'https://api.opendota.com/api/publicMatches' )
4      mcontent = mr.content.decode('utf-8')
5      md = json.loads(mcontent)
6      if type(md) is str:
7          continue
8
9      for match in md:
10         if type(match) is str:
11             continue
12         time.sleep(1)
13         mmr = requests.get( 'https://api.opendota.com/api/matches/' + str(match['match_id']))
14         mmcontent = mmr.content.decode('utf-8')
15         mmd = json.loads(mmcontent)
16
17         if 'region' in mmd.keys():
18             if mmd['region'] in [1,2] and mmd['match_id'] not in match_ids and mmd['chat'] is not None:
19                 match_ids.append(mmd['match_id'])
```

Which will use the Dota 2 API to get a list of 10 random matches, and add to *match_ids* those matches from the proper regions, who have not yet been added to the list, and contain at least one chat message. We then use the API to access the complete data for each suitable match (line 13 above). The *continue* in case of *str* type objects prevents API overuse error messages being added to *match_ids*. This rudimentary error handling could certainly be improved.

3. Message Classification

We use the Google Perspective API to classify each message in a match as toxic or non-toxic. This service provides four different axes along which to classify comments, and all comments were analyzed according to each criteria.

For each match we create a list for each Perspective parameter. We then loop through the messages for each parameter, and for each message create list with the message text, the Perspective API score, and the slot of the sending player. Each of these lists is added to the appropriate list, according to which perspective parameter was used in classification. We then create a dictionary *target*, which has as keys player slots, and as values 0 or 1, depending on whether or not that player sent a toxic message. See line 158:

```python
for res in results_default:
        if res[2] not in target.keys():
            target[res[2]] = 0
        if res[1] > .7:
            target[res[2]] = 1
```

To use a different Perspective parameter, change *results_default* to the appropriate list. To adjust the threshold for concluding toxic messages were sent, change the value .7 to your desired sensitivity. (Google Perspective API returns a value between 0 and 1, 1 being most sure of toxicity.) These *target* dictionary objects are then placed into a dictionary called *targets_dict*, which uses match ID's as keys, and the associated *target* dictionary as values.

To summarize, this process allows us to identify whether or not the player in slot s, in match m, sent a toxic message or not by calling:

```python
targets_dict[m][s]
```

where m and s are of type *int*

4. Data Processing

4.1. Events List

Beginning on line 736, we iterate through each match whose data we have downloaded, and initialize a list e, which we will populate with events that took place in the match, particularly those in the list *logs*, initialized at the beginning of the script. This occurs at line 763:

```python
    for player_arr in matches[match_id]['players']:
        if player_arr['player_slot'] is not None:
            iden = str(player_arr['player_slot'])
            for log in logs:
                event_type = log
                if player_arr[log] is None:
                    continue
                for item in player_arr[log]:
                    e.append([item, iden, event_type])
```

Due to later processing, it is crucial that each element *event* of e have the above structure, where $event[0]$ is the event as a dictionary containing, at the very least, the time at which the event occurred, $event[1]$ contains the slot of the player involved in the event, and $event[2]$ designates what type of event occurred.

4.2. Deaths Log

The Dota 2 API does not provide a log of each player's deaths, useful as this seems. So it is necessary to construct a deaths log, given each kill in the game. This occurs at line 739:

```python
 for player_arr in matches[match_id]['players']:
        isR = player_arr['isRadiant']
        if player_arr['kills_log']:
            for kill in player_arr['kills_log']:
                t = kill['time']
                hero = kill['key'][14::]

                for hero_id in hero_id_list:
                    if hero_id['name'] == hero:
                        Id = hero_id['id']
                        for player_arr2 in matches[match_id]['players']:
                            if player_arr2['hero_id'] == Id and player_arr2['isRadiant'] != isR:
                                hero = player_arr2['player_slot']
                e.append([kill, hero, 'death_log'])
```

For each player in a given match, we loop through their *kills_log* (if it is non-empty, i.e if this player had a kill in the match), and record the time of the kill, and which hero was killed. A quirk in the Dota 2 API requires we use the *hero_id_dict*, defined just above this code block, to identify the *hero_id* of the killed player, then loop through the list of players to find the player with that *hero_id*, on the other team than the player who recorded the kill. This process could be substantially sped up through the use of appropriate dictionaries. Once we have identified the player that was killed, we append to e a death event with the required structure.

4.3. Time Divisions

Each match has a different length, and the Dota 2 API has a to-the-second fidelity. So, the most faithful representation of match events would be a second by second log of which events occurred. However, a record of normalized length is desirable so that machine learning techniques may be applied. We opt here to record which events occurred in each percentile of the match. On line 774 we have:

```python
times_dict = {}

    for i in range(102):
        times_dict[i]= i* (matches[match_id]['duration']//100)
```

This *times_dict* records which time in the match corresponds to each percentile.

4.4. Instances

Beginning at line 780, we construct a a dictionary for each event type, for each player, whose keys are percentiles of the match duration, and whose values are 0 or 1, depending on whether or not the player produced that event at that point in the match.

```python
    for item in e:
        cols.append(str(item[1]) + str(item[2]))

    cols = set(cols)

    d_list = []

    for col in cols:
        dic = {'name': col}
        d_list.append(dic)



    for dic in d_list:
        for t in range(101):
            dic[t] = 0


    for event in e:
        for dic in d_list:
            if str(event[1])+str(event[2])==dic['name']:
                for i in range(101):
                    if times_dict[i] != 'name' and times_dict[i+1]!= 'name':
                        if times_dict[i] <= event[0]['time'] < times_dict[i+1]:
                            dic[i] = 1
```

We then add this dictionary to a dictionary *dfs_dict* whose keys are match IDs. To summarize, we find the normalized time series for the event type e, in match m, for the player in slot s by calling

```python
dfs_dict[m][s+e]
```

where $s + e$ is the string concatenation of s and e. We now wish to produce instances, i.e rows who contain the normalized time series representation of each event type, for each player in each match. On line 819, we have:

```python
dfs_match_player = {}

for key in dfs_dict.keys():
    dfs_match_player[key] = {}
    for i in range(0, 257):
        df = pd.DataFrame()
        for col in dfs_dict[key].columns:
            c = re.sub("\D", "", col)
            if c == str(i):
                f=dfs_dict[key]
                df[col] = f[col]
        if not df.empty:
            dfs_match_player[key][i] = df
```

So that we may find the player p in match m's normalized time series for each event as columns by running

```python
dfs_match_player[m][p]
```

What remains is to turn this data frame into it's long form and regularize column names, accomplished at line 833:

```python
instances = []

for matchkey in dfs_match_player.keys():
    for playerkey in dfs_match_player[matchkey].keys():
        work_df = dfs_match_player[matchkey][playerkey]
        work_df = work_df.iloc[1:]
        work_df = work_df.stack().to_frame().T
        work_df.columns = ['{}_{}'.format(*c) for c in work_df.columns]

        if playerkey in targets_dict[matchkey].keys():
            work_df['target'] = targets_dict[matchkey][playerkey]
            instances.append(work_df)
```

At this stage, we may concatenate all of these instances into a data frame by running:

```python
cols_ideal = []


for df in instances:
    df.rename(columns=lambda x: x.split('_',1)[0] + ''.join([i for i in x.split('_',1)[1] if not
    i.isdigit()]) if x != 'target' and type(x) is not int else x, inplace=True)
    for col in df.columns:
        cols_ideal.append(col)

cols_ideal = set(cols_ideal)

vs = pd.DataFrame()

for df in im_list:
    if set(df.columns) != set(cols_ideal):
        for c in list(set(cols_ideal)-set(df.columns)):
            df[c] = 0
    df.reset_index(drop=True, inplace=True)

vs = pd.concat(instances, axis = 0, sort = False)
```

4.5. Meta-Statistics

Data frames of the above structure were not useful in our investigation, so we continue to create meta-statistics, considering which events occurred in proximity to others. On line 847 we have:

```python
instances_modded=instances

cats = logs
im_list = []

for instance in instances_modded:
    instance['death_buybacks'] = 0
    for i in range(100):
        if int(instance[str(i)+'death_log'] ) == 1:
            for j in range(5):
                try:
                    if int(instance[str(i+j)+'buyback_log']) == 1:
                        instance['death_buybacks'] += 1
                except Exception as e:
                    print(e)
```

This creates a factor *death_buybacks*, which identifies how many times a player used a buyback in close proximity to death, a behavior associated with frustration in Dota 2. In this case the proximity is 5% of the match, which clearly may be modified. This section of code may also be expanded to consider the proximity between other events. Now we examine events of the same type occurring in close proximity:

```python
for instance in instances_modded:
    instance_meta = pd.DataFrame()
    for cat in cats:
        print(cat)
        columns_rel = [col for col in instance.columns if cat in col]
        columns_rel.sort(key =lambda column: non_decimal.sub('', column))
        for i in range(2,5):
            print(i)
            instance_meta[cat+str(i)] = [0]
            for j in range(96):
                print(j)
                check_cols = columns_rel[j:j+i]
                if int(instance[check_cols].sum(axis = 1)) == i:
                    instance_meta[cat + str(i)] = [1]
        instance_meta[cat+'total'] = int(instance[columns_rel].sum(axis = 1))
    instance_meta['target'] = instance['target']
    instance_meta['death_buybacks'] = instance['death_buybacks']
    im_list.append(instance_meta)
```

For each event type, *instance_meta* records if this event occurred twice, three times, or four times in close proximity. Again we consider close proximity to be within 5% of the match duration of each other, which may be modified. Finally, we condense this list of meta-instances into a data frame via:

```python
cols_ideal = []


for df in instances_modded:
    df.rename(columns=lambda x: x.split('_',1)[0] + ''.join([i for i in x.split('_',1)[1] if not i.isdi
    for col in df.columns:
        cols_ideal.append(col)

cols_ideal = set(cols_ideal)

vs = pd.DataFrame()

for df in im_list:
    if set(df.columns) != set(cols_ideal):
        for c in list(set(cols_ideal)-set(df.columns)):
            df[c] = 0
    df.reset_index(drop=True, inplace=True)

vs = pd.concat(instances_modded, axis = 0, sort = False)
```

5. Results

We split the 637 instances into train and test data, using 15% of instances as test data. We use skLearn's RandomForest Regressor, with 5000 estimators. This model fails to predict a single instance of toxic behavior, that is, for each instance it predicts that no toxic behavior occurred. One possible explanation is low occurrence of toxic behaviors. Out of 637 instances, only 42 contain a toxic behavior as we have classified it. Lowering the threshold we use with the Perspective API to conclude toxicity could increase our capture of toxic behaviors, but I don't believe this is a viable solution as we already classify some messages as toxic that many may not consider truly so. A more targeted match selection, and increased sample size seem better remedies. Another problem is the relevance of our predictors. Consider the cross-tabulation of a factor and our target variable. Relevant predictors will display a low proportion of false negatives and false positives. Many of our predictors have this property, for example, below we consider *kills_log*3:

```
pd.crosstab(vs['target'],vs['kills_log3'])

out:
kills_log3    0   1
target
0           575  20
1            41   1
```

However, we lack true positives. Examining true positives for all predictors which take binary values, we see we have the most true positives with *purchase_log*2:

```
for column in vs.columns:
    if column is not 'target':
        c = pd.crosstab(vs['target'],vs[column])
        if c.shape == (2,2) :
            l.append((c[1][1], column))
print(max(l))

out:
(40, 'purchase_log2')
```

But closer inspection gives:

```
pd.crosstab(vs['target'],vs['purchase_log2'])

out:
purchase_log2   0    1
target
0              43  552
1               2   40
```

That is, the proportion of true positives and true negatives is nearly equal, and the proportion of false positives is very high, making this predictor non-viable.

6. Conclusions and Extensions

Due to low factor relevancy and low occurrence of toxic behaviors (as classified in Section 3) we were unable to produce a model that correctly predicts instances of toxic behavior based on in-game actions. However we do provide a data-pipeline that will, given match data from the Dota 2 API, produce either a normalized time series depiction of the match's events, or provide meta-statistics describing the match. Future investigators will be able to modify the classification of toxic behaviors, as well as consider a different set of match events, and change the time divisions in order to continue investigation. They may also expand and modify the collection of meta statistics to produce more relevant predictors.

7. Using this Script

Using Spyder, the variables (and as such, the collected data so far) were exported as *vars.spydata*. These variables may be imported in the Variable Explorer pane. I do not know if this is possible using a different IDE. Using this data will save hours of runtime. If you do import these variables, do not run the script as a whole, as they will be overwritten as empty lists at the beginning of the program. Run only the sections you need.

I have removed my Google API key, so you will need to use your own. You can get one here, and it must be inserted at line 116.

Most of this code was written in a data-sciencey, "shoot from the hip," style. I hope this documentation will allow it to be understood and modified in a productive way, but if a section is insufficient or unclear, feel free to email cflaherty@middlebury.edu.