The final task

Subject area: Dota 2 game

Dota 2 (https://ru.wikipedia.org/wiki/Dota_2) is a multiplayer computer game of the MOBA (https://ru.wikipedia.org/wiki/MOBA) genre . Players play matches against each other. Each match involves two teams, 5 people each. One team plays for the light side (The Radiant), the other for the dark side (The Dire). The goal of each team is to destroy the main building of the enemy base (the throne).

There are different game modes (http://dota2.gamepedia.com/Game_modes/ru) , we will be looking at Captain's Mode (http://dota2.gamepedia.com/Game_modes/ru#Captain.27s_Mode) , which is the format in which most Dota 2 eSports events take place.

How is the match going?

1. Players choose heroes

There are just over 100 different heroes (characters) in the game. At the beginning of the game, teams select heroes for themselves in a certain order and prohibit the opponent from selecting certain heroes (bans). Each player will control one hero; there cannot be several identical heroes in one match. Heroes differ from each other in their characteristics and abilities. The success of the team largely depends on the combination of heroes selected.

2. Main part

Players can earn gold and experience for killing other heroes or other units. The accumulated experience affects the hero's level, which in turn allows you to improve abilities. For the accumulated gold, players buy items that improve the heroes' characteristics or give them new abilities.

After death, the hero goes to the "tavern" and is revived only after some time has passed, so the team loses the player for some time, but the player can buy the hero out of the tavern early for a certain amount of gold.

During the game, teams develop their heroes, defend their part of the field and attack the enemy.

3. End of the game

The game ends when one team destroys a certain number of the opponent's "towers" and destroys the throne.

Task: predicting victory based on data from the first 5 minutes of the game

Based on the first 5 minutes of the game, predict which team will win: Radiant or Dire?

Dataset

The match data set is written in the file `matches.jsonlines.bz2` . The catalog `dictionaries` contains the decoding of the identifiers that are present in the match records.

Reading match information

Match information is stored in a compressed text file `matches.jsonlines.bz2` , each line of which contains a JSON (https://ru.wikipedia.org/wiki/JSON) object . The JSON record is converted to a Python object using the standard module `json` . An example of reading matches:

In [5…

```
import  json
import  bz2

with  bz2 . BZ2File ( './matches.jsonlines.bz2' )  as  matches_file :
    for  line  in  matches_file :
        match  =  json . loads ( line )

        # Handling match
        break
```

Description of fields in the match record

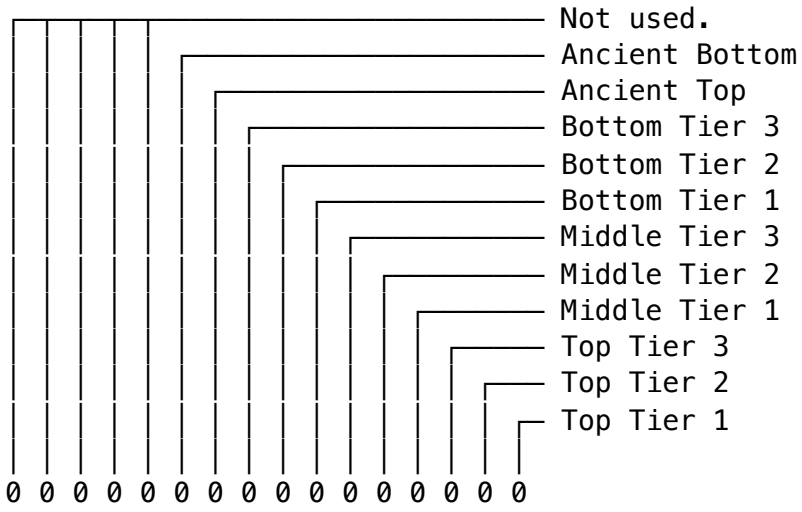
```
{
    "match_id" :  247 ,             # match identifier
    "start_time" :  1430514316 ,    # match start date/time, unixtime
    "lobby_type" :  0 ,             # type of room in which players gath
er
                                # (explanation in dictionaries/lobbies.c
sv)

    # hero selection stage
    "picks_bans" :  [
        {
            "order" :  0 ,        # action ordinal number
            "is_pick" :  false ,  # true if the team picks the hero, fal
se if it bans
            "team" :  1 ,         # team performing the action (0 — Radi
ant, 1 — Dire)
            "hero_id" :  95      # hero associated with the action
                             # (explanation in dictionaries/heroes.csv)
        },
        ...
    ],

    # information about each player, a list of exactly 10 elements
    # players with indexes from 0 to 4 are from the Radiant team, from 5
to 9 are Dire
    "players" :  [
        {

            # player hero (explanation in dictionaries/heroes.csv)
            "hero_id" :  67 ,

            # time series (counts are specified in the "times" field)
            "xp_t" :  [ 0 ,  13 ,  115 ,  177 ,  335 ,  ... ],    # expe
rience
            "gold_t" :  [ 0 ,  99 ,  243 ,  343 ,  499 ,  ... ],  # gold
+ the cost of all purchased items (net worth)
            "lh_t" :  [ 0 ,  0 ,  2 , 2 ,  2 , ... ], # the number of en
emy units (not heroes) killed

            # list of events: hero ability upgrades
            "ability_upgrades" :  [
                {
                    "time" :  51 ,       # game time
                    "level" :  1 ,       # player level at which upgrade
occurred
                    "ability" :  5334   # ability that was upgraded
                                    # (decrypted in dictionaries/abilit
ies.csv)
                },
                ...
            ],

            # list of events: kills
```

```
            "kills_log" :  [
                {
                    "time" :  831 ,     # game time
                    "player" :  7 ,     # index of the player whose hero
was killed
                                        # (blank if a non-hero was killed)
                    "unit" :  "npc_dota_hero_viper"  # type of killed un
it
                },
                ...
            ],

            # event list: item purchases
            "purchase_log" :  [
                {
                    "time" :  -73 ,      # game time
                                         # the game time reference point (ze
ro) starts
                                         # a few minutes after the actual st
art of the match, so
                                         # some event times may be negative
                    "item_id" :  44     # item purchased (decrypted in d
ictionaries/items.csv)
                },
                ...
            ]

            # list of events: hero buyback from tavern
            "buyback_log" :  [
                { "time" :  2507 },
                ...
            ],

            # list of events: hero sets up "observers" that allow the te
am
            to # monitor part of the playing field at some distance from
the point of setting
            "obs_log" :  [
                {
                    "time" :  1711 ,     # game time of setting
                    "xy" :  [ 111 ,  130 ]  # coordinates of the playing
field
                },
                ...
            ],
            "sen_log" :  [],  # similar to the obs_log field, another ty
pe of "observer"

        },
        ...
    ],

    # game time counts at which the values of the time series
```

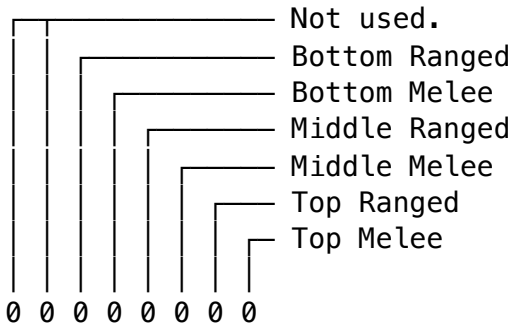
```
    "times" are calculated :  [ 0 ,  60 ,  120 ,  180 ,  ... ],

    # key game events
    "objectives" :  [
        {
            "time" :  198 ,            # event time
            "type" :  "firstblood" ,   # event type
            "player1" :  6 ,           # event parameters, may contain
            "player2" :  1           # player indices (player),
                                  # team number (team, 0 — Radiant, 1 —
Dire)
        },
        {
            "time" :  765 ,
            "type" :  "tower_kill" ,
            "player" :  7 ,
            "team" :  1
        },
        ...
    ]

    # match result (missing in test matches)
    "finish" :  {
        "duration" :  2980 ,             # duration in seconds
        "radiant_win" :  false ,         # true if the Radiant team won
        "tower_status_radiant" :  0 ,    # state of the teams' towers a
t the end of the game
        "tower_status_dire" :  1972 ,    # (see bitmask description)
        "barracks_status_dire" :  63 ,   # state of the teams' barracks
at the end of the game
        "barracks_status_radiant" :  0   # (see bitmask description)
    }
}
```

Description of the status fields of towers and barracks

The state of the towers at the end of the game is given by an integer, encoded in bits:



The state of the barracks at the end of the game is encoded in integer bits:



Feature extraction

The extract_features.py script extracts features from known match information for the first 5 game minutes and creates a table from them. The table will help you quickly form an object-feature matrix, a response vector, and start applying machine learning methods to solve the problem.

The features presented in the table `features.csv` are considered by experts in the subject area to be the most important for solving the problem of predicting a team's victory. However, it is not necessary to use these features in their original form to apply machine learning methods - you can create new features from the existing ones. Moreover, the features in the file `features.csv` do not contain all the information known about the match for the first 5 game minutes. You can use the script `extract_features.py` as an example and add your own features to improve the quality of the prediction.

Example of reading a file with features

In [3…
```
import  pandas
features  =  pandas . read_csv ( './features.csv' ,  index_col = 'match_id'

features . head ()
```

Out[3]:

match_id	start_time	lobby_type	r1_hero	r1_level	r1_xp	r1_gold	r1_lh	r1_kills	r1_deaths	r
0	1430198770	7	11	5	2098	1489	20	0	0	
1	1430220345	0	42	4	1188	1033	9	0	1	
2	1430227081	7	33	4	1319	1270	22	0	0	
3	1430263531	1	29	4	1779	1056	14	0	0	
4	1430282290	7	13	4	1431	1090	8	1	0	

5 rows x 108 columns

Description of features in the table

- `match_id` : match id in the dataset
- `start_time` : match start time (unixtime)
- `lobby_type` : the type of room in which players gather (explanation in `dictionaries/lobbies.csv`)
- Sets of attributes for each player (players of the Radiant team - prefix `rN` , Dire - `dN`):
  - `r1_hero` : player hero (explanation in dictionaries/heroes.csv)
  - `r1_level` : maximum level achieved by the hero (in the first 5 game minutes)
  - `r1_xp` : maximum experience gained
  - `r1_gold` : achieved hero value
  - `r1_lh` : number of units killed
  - `r1_kills` : number of killed players
  - `r1_deaths` : number of hero deaths
  - `r1_items` : number of items purchased
- Signs of the event "first blood". If the event "first blood" did not happen in the first 5 minutes, then the signs take the missing value
  - `first_blood_time` : first blood game time
  - `first_blood_team` : team that scored first blood (0 - Radiant, 1 - Dire)
  - `first_blood_player1` : a player involved in an event
  - `first_blood_player2` : the second player involved in an event
- Features for each command (prefixes `radiant_` and `dire_`)
  - `radiant_bottle_time` : the time the team first acquired the item "bottle"
  - `radiant_courier_time` : time of acquisition of the item "courier"
  - `radiant_flying_courier_time` : time of acquisition of item "flying_courier"
  - `radiant_tpscroll_count` : number of "tpscroll" items in the first 5 minutes
  - `radiant_boots_count` : number of items "boots"
  - `radiant_ward_observer_count` : number of items "ward_observer"
  - `radiant_ward_sentry_count` : number of items "ward_sentry"
  - `radiant_first_ward_time` : the time the team installs the first "observer", i.e. an object that allows you to see part of the playing field
- Match summary (these fields are not included in the test sample because they contain information beyond the first 5 minutes of the match)
  - `duration` : duration
  - `radiant_win` : 1 if Team Radiant won, 0 otherwise
  - The state of towers and barracks at the end of the match (see description of dataset fields)
    - `tower_status_radiant`
    - `tower_status_dire`
    - `barracks_status_radiant`
    - `barracks_status_dire`

Quality metric

As a quality metric, we will use the area under the ROC curve (AUC-ROC). Note that AUC-ROC is a quality metric for the algorithm that produces estimates of membership in the first class. Both algorithms that will be used in the project - gradient boosting and logistic regression - can produce such estimates. To do this, you need to get predictions using the predict_proba function. It returns two columns: the first contains estimates of membership in the zero class, the second - in the first class. You need the values from the second column:

```
pred  =  clf . predict_proba ( X_test )[:,  1 ]
```

Solution Guide

You need to complete the two stages of research described below, write a short report based on the results of each stage (the questions that should be answered in the report are listed below), and provide this report and the code with which you completed the task for review.

Please note: high quality of cross-validation work (close to 100%) is primarily a reason to think about whether you are training the model correctly. Perhaps you are looking into the future or tuning on the wrong set of features.

Approach 1: Head-on gradient boosting

One of the most versatile algorithms studied in our course is gradient boosting. It is not very demanding on data, restores nonlinear dependencies, and works well on many data sets, which explains its popularity. It would be a reasonable idea to try it first.

1. Read the features table from the features.csv file using the code above. Remove the features related to match results (they are marked in the data description as missing from the test sample).
2. Check the sample for gaps using the count() function, which shows the number of filled values for each column. Are there many gaps in the data? Write down the names of the features that have gaps, and try to justify why their values may be missing for any two of them.
3. Replace gaps with zeros using the fillna() function. This is actually the preferred method for logistic regression, as it will allow the missing value to contribute nothing to the prediction. For trees, it is often best to replace a gap with a very large or very small value - in this case, when constructing a node split, you can send features with gaps to a separate branch of the tree. There are also other approaches - for example, replacing a gap with the mean value of the feature. We do not require this in the task, but if you want, try different approaches to handling gaps and compare them with each other.
4. Which column contains the target variable? Write down its name.
5. Let's forget that there are categorical features in the sample and try to train gradient boosting over trees on the existing "objects-features" matrix. Fix the split generator for

cross-validation by 5 blocks (KFold), do not forget to shuffle the sample (shuffle=True), since the data in the table is sorted by time, and without shuffling you can encounter undesirable effects when assessing the quality. Evaluate the quality of gradient boosting (GradientBoostingClassifier) using this cross-validation, try different numbers of trees (at least test the following values for the number of trees: 10, 20, 30). How long did it take to tune the classifiers? Is the optimum reached with the tested values of the n_estimators parameter, or is the quality likely to continue to grow with its further increase?

What to include in the report

In your report for this stage, you should answer the following questions:

1. Which features have gaps in their values? What might gaps in these features mean (answer this question for any two features)?
2. What is the name of the column that contains the target variable?
3. How long did it take to cross-validate gradient boosting with 30 trees? The instructions for measuring time can be found below. What was the quality? Recall that in this task we use the AUC-ROC quality metric.
4. Does it make sense to use more than 30 trees in gradient boosting? What would you suggest to do to speed up its training as the number of trees increases?

Recommendations and tips

- If everything is running very slowly:
  - Use not the entire sample for training and cross-validation, but a subset of it - for example, half of the objects. It is best to take the subset randomly, rather than forming it from the first m objects.
  - Try simplifying the model - for example, reducing the depth of trees in gradient boosting (max_depth).

Measuring code execution time

```
import  time
import  datetime

start_time  =  datetime . datetime . now ()

time . sleep ( 3 )  # place the measured code instead of this line

print  'Time elapsed:' ,  datetime . datetime . now ()  -  start_time
```

Approach 2: Logistic Regression

Linear methods are much faster than tree compositions, so it seems reasonable to use them to speed up data analysis. One of the most common methods for classification is logistic regression.

Important: don't forget that linear algorithms are sensitive to feature scale! sklearn.preprocessing.StandartScaler may be useful.

1. Evaluate the quality of logistic regression (sklearn.linear_model.LogisticRegression with L2 regularization) using cross-validation using the same scheme used for gradient boosting. Find the best regularization parameter (C). What is the best quality you got? How does it compare to the quality of gradient boosting? How can you explain this difference? Is logistic regression faster than gradient boosting?
2. Among the features in the sample, there are categorical ones, which we used as numeric ones, which is hardly a good idea. There are eleven categorical features in this problem: lobby_type and r1_hero, r2_hero, ..., r5_hero, d1_hero, d2_hero, ..., d5_hero. Remove them from the sample and conduct cross-validation for logistic regression on a new sample with the selection of the best regularization parameter. Has the quality changed? How can you explain this?
3. In the previous step, we excluded the rM_hero and dM_hero features from the sample, which show which heroes played for each team. These are important features - heroes have different characteristics, and some of them win more often than others. Find out from the data how many different hero IDs exist in a given game (you may need the unique or value_counts function).
4. Let's use the "bag of words" approach to encode information about heroes. Let's say there are N different heroes in the game. Let's form N features, where the i-th will be equal to zero if the i-th hero did not participate in the match; one if the i-th hero played for the Radiant team; minus one if the i-th hero played for the Dire team. Below you can find the code that performs this transformation. Add the resulting features to the numerical ones that you used in the second point of this step.
5. Conduct cross-validation for logistic regression on a new sample with the best regularization parameter. What is the quality? Has it improved? How can you explain this?
6. Build predictions of the Radiant team's win probabilities for the test set using the best of the studied models (the best in terms of AUC-ROC on cross-validation). Make sure that the predicted probabilities are adequate - they are on the interval [0, 1], do not coincide with each other (i.e. that the model is not constant).

What to include in the report

In your report for this stage, you should answer the following questions:

1. What is the quality of logistic regression over all original features? How does it compare to the quality of gradient boosting? How can you explain this difference? Is logistic regression faster than gradient boosting?
2. How does removing categorical features (specify the new value of the quality metric) affect the quality of logistic regression? How can you explain this change?
3. How many different hero IDs are there in this game?
4. What was the quality like when adding the "bag of words" for the characters? Did it improve compared to the previous version? How can you explain this?
5. What is the minimum and maximum forecast value on the test sample obtained by the best algorithm?

Code for forming a "bag of words" by characters

```
# N — number of different heroes in the sample
X_pick  =  np . zeros (( data . shape [ 0 ],  N ))

for  i ,  match_id  in  enumerate ( data . index ):
    for  p  in  xrange ( 5 ):
        X_pick [ i ,  data . ix [ match_id ,  'r %d _hero'  %  ( p + 1
)] - 1 ]  =  1
        X_pick [ i ,  data . ix [ match_id ,  'd %d _hero'  %  ( p + 1
)] - 1 ]  =  - 1
```

Testing the final model

After you have run all the experiments and selected the best model, you can check its quality in test matches. The sample of test matches is collected in the file `matches_test.jsonlines.bz2` . Unlike the main set of matches, test matches only contain information that is known at the time of the first 5 game minutes, the result of the match is unknown. The feature table for test matches is `features_test.csv` .

For all matches in the test set, predict the probability of Radiant winning, write the predictions to a CSV file with columns `match_id` (match ID) and `radiant_win` — predicted probability. The file with predictions should look something like this:

```
match_id,radiant_win
1,0.51997370502
4,0.51997370502
15,0.51997370502
...
```

Submit your solution to Kaggle competition: Dota 2: Win Probability Prediction.

Competition Link: Dota 2: Win Probability Prediction
(https://kaggle.com/join/coursera_ml_dota2_contest)

What else to try?

Of course, there are many more ideas you can try that will help you get an even higher score on kaggle. Here are just a few possible options:

1. There are quite a lot of indicators for each player: maximum experience, number of deaths, etc. (see the list above). You can try to sum them up or average them, getting aggregated indicators for the entire team.
2. The raw data (file matches.jsonlines.bz2) contains a lot of information that we haven't used yet. You could, for example, create "bags of words" for purchases of different items (i.e. encode information about how many times each team purchased a particular item). Note that this can result in too many features, for which it might make sense to do dimensionality reduction using principal component analysis.

3. It is possible to generate features about changes in heroes' abilities during the match (ability_upgrades).
4. This task only uses gradient boosting and logistic regression - but we studied other models too! You can try k-nearest neighbors, SVM, random forest, etc.

About the task and the final assignment

Why this particular task?

- Publishing real data from industrial tasks is a very bold step for a company. Few (even Yandex) can do this. It is much easier (and sometimes more interesting) to use data from open sources.
- Public datasets from the Internet are of little use for solving real business problems, which is why they are in the public domain.
- We chose to do a toy problem on real data instead of a real problem on toy data.
- The task of predicting victory is a toy, but here is just a small list of real-life tasks that it resembles:
  - Predicting the probability of a bank client purchasing a service
  - Predicting the likelihood of customer defection to another service provider
  - ... (think of other examples)

The task is too simple. What else can be done?

Answer the question: what is the minimum number of minutes of a match that you need to know in order to correctly guess the winning side in 80% of matches? And with 90% accuracy? Give your answer to this question and prove that such accuracy can really be achieved by building a model and qualitatively validating it. How predictable are matches in the game Dota 2?

Write an article about it, tell everyone, and come to us for an interview.

Where did you get the data?

The dataset was made based on the YASP 3.5 Million Data Dump (http://academictorrents.com/details/5c5deeb6cfe1c944044367d2e7465fd8bd2f4acf) of Dota 2 match replays from yasp.co (http://yasp.co/) . Many thanks to Albert Cui, Howard Chung and Nicholas Hanson-Holtry for the dump. The dump is licensed under CC BY-SA 4.0.

How was the sample formed?

The original match upload has been cleared, the proposed set contains the following matches:

- played from 2015-05-01 to 2015-12-17
- lasting at least 15 minutes
- matches with incomplete information have been removed (for example: information about players is missing)

From the entire dataset, 15% of random records were selected as a test set.

In order to discourage Kaggle competitors from taking high places using cheating methods (for example, by downloading the original data set and looking at the answers on the test set of matches), we performed minimal data obfuscation, i.e., slightly confused the dataset:

- changed match identifiers
- the start time of each match was shifted by the value of a random variable, normally distributed with a standard deviation of 1 day