Topcoder Marathon Match Report
FishingForFishermen
by Christoph Langer, MSc

First name: Christoph
Last name: Langer
Topcoder handle: chlanger
Email address: christophlanger@protonmail.com

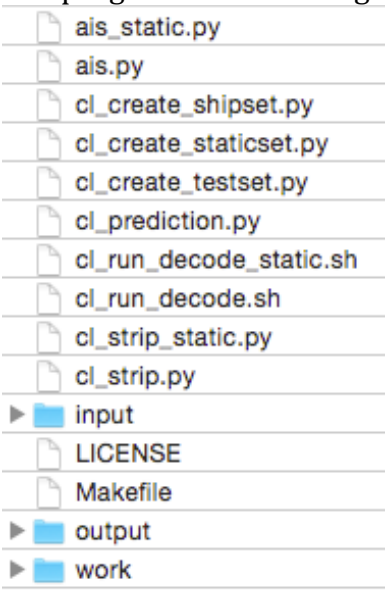
General remarks on the contest:

I very much enjoyed the format of this contest. I was a bit skeptical about the decoding part of the contest at first but feature extraction is a great part of data science and it forces you to look at your data in more detail. Further this allows more and better educated guesses and makes the competition less about hyper parameter tuning. The narrow time window for this competition and not being able to work on it fulltime, forced me to pair down my ideas on the ones which seemed the most promising instead of being able to perform exhaustive trial and error searches. This left me with sleek and simple model. Since I had a very finite number of submissions left, I could only use the public scoreboard as a sanity check and relied on my internal testing.

All of the code was run on an old 2.53 GHz Intel Core 2 Duo MacBook Pro with 8GB RAM. I used a xgboost build without multi-threading, because by default clang in OSX does not come with open-mp, as mentioned in:
https://github.com/dmlc/xgboost/blob/master/doc/build.md

Quick information how to use the program:

The program has following file structure:



Put the `static_reports_raw_corrected.csv, testing_data.csv, training_data.csv` into the input folder.

Run `'make decode'` - this will decode `static_reports_raw_corrected.csv` and `testing_data.csv` and generate all data sets needed for the prediction into the work folder, intermediate files not needed for the model are zipped and could be deleted.

Run `'make decode'` – this will generate the model and save the predictions into the output folder.

Attention: On the Linux Server .sh-scripts had to be called with the command bash instead of sh (This happens int the Makefile).

Approaches considered:

One of the reasons I was driven to this contests was the point:

„Your algorithm shall utilize: Other data from openly accessible data sources that can be used to determine fishing behavior."

This was my chance to evade the model-tuning rat race and focus more on clever feature extraction and engineering. I wanted to include following additional data:

Bathymetry: A vessel is usually only equipped to fish at certain depth. It is presumably only crossing depth not fitting its' profile. I started working on this using the data of the NOAA - ETOPO1 Global Relief Model but had to cut it because of time constraints.

***Wave Height Data*:** Depending on the vessel, wave height limits the general ability to fish at this particular sea conditions. Again cut due to time constrains.

Marine protected areas: Although this would have probably increased the overall accuracy of my algorithm, it felt counterproductive to the goal of this competition thus to detect illegal fishing behavior.

Approach ultimately chosen:

Data used: I used the train and test sets provided in this competition, as well as the static data set. The original static set provided had a missing column and inmessage repeats. However the solution submitted uses the new static set provided in which this has been corrected.

From the raw messages in `testing_data.csv` the same fields as in `training_data.csv` were extracted. From the `static_reports_raw_corrected.csv` I extracted all fields available in the messages. I then extracted vessel characteristics into a new file. I grouped the static reports by ship using the `'MMSI'.` However I only used `'SHIPTYPE', 'DRAUGHT'` and one engineered feature: `'TO_BOW'+'TO_STERN` (=Overall ship length) in the model.

My reasoning was that hull depth and overall ships length are the most relevant features out of the ones available to determine a boats performance and characteristics.

This boat specific data was merged onto the train and test set where available.

I extracted from the `'TIMESTAMP'` following addition features: `'year', 'month', 'day', 'weekday'` and `'weekend'`. `'TIMESTAMP'` was also kept as running time measurement.

All Categorical features were one hot encoded.

Model used: I am using `XGBoost` (http://xgboost.readthedocs.io/en/latest/model.html) a Boosted Tree model using Gradient Boosting. The hyper parameters were tuned using working parameters from a previous project and started iterating around them using `LabelShuffleSplit` based on `'MMSI'` (and using different random seeds) as a validation tool.

The tree approach was chosen in particular because it handles coordinates intrinsically, no distance calculations are necessary.

Open source resources and tools used, including URLs and references to specific lines of your code:

References to specific lines in the code are given in later segments of this report.

The information provided in the competition about AIVDM/AIVDO protocol decoding, is the document for the GPSD project and was developed to be the specification for it. Thus it felt only natural to used from the ***GPSD project*** (**http://catb.org/gpsd/**), (release release-3.16.tar.gz) the standalone decoder found at: /devtools/ais.py.

License: BSD

I am using the ***Anaconda*** distribution of the Python:
https://www.continuum.io/downloads

It includes following useful python libraries:

Pandas - http://pandas.pydata.org/

Numpy - http://www.numpy.org/
Sklearn - http://scikit-learn.org/stable/
All of licensed under BSD.

XGBoost Library:
https://xgboost.readthedocs.io/en/latest/
License:
Apache License, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0)

Comments on libraries:
The default OSX installation of XGBoost runs in single threaded mode, previous work with the library, has shown that multicore execution gives similar but not identical solutions.
Anaconda, XGBoost where used unchanged ***the GPSD decoder*** was slightly altered to decode the static data (line 881 ff.). By increasing the length ranges of messages, thus accounting for the padding in the transmissions.

Comments on open source resources used:
As mentioned previously due to time constrains no open source data resources where used. This submission exclusively uses the data provided in the contest.

Steps to approach ultimately chosen, including references to specific lines of your code:
The code is split into multiple Python and Shell scripts, however they are tied together by a easy to use Makefile. **`make info:`** Will tell you where to put to input data and naming convention. **`make decode`**: Will decode the AIS messages, extract all features and save them in .csv files. At the end it will compress all files, which may hold information for future endeavors but are not needed in the final model. **`make predict:`** will generated the model and output the predicted set . None of the code needs to compile, since it is written exclusively in python and shell scripts working together with the libraries mentioned above.

Now an in depth looks what my scripts do:

Decoding and preparing the test set:

- `cl_strip.py`
  Creates a file `test_stripped.csv` into the /work directory. It basically creates a file with AIVDM messages lines after line (= 'RAW_MESSAGE' field without quotes) which is feed to the GPSD standalone decoder in the next script. In addition I realized some 'RAW_MESSAGE' fields where missing the '!' in front of the AIVDM tag. This is corrected in line 15ff and the script outputs to the standard output, how often this has happened.
- `cl_run_decode.sh`
  Is a simple shell script that runs the unmodified ais.py stand-alone decoder (in Python 2.7). Input is the previously created `test_stripped.csv` and outputs `test_decoded.txt`. ais.py is run with –jm flags, ensuring that

every line is either a string with is the decoded message in JASON format, or a dump of the 'RAW_MESSAGE' field if malformed.

- `cl_create_testset.py`
  Creates a file testing `testing_data.csv,` which is of identical makeup (identical columns) as the provided `training_data.csv`.
  In line 22ff it creates `'TIME',` out of the unix timestamp. This is done only for completeness' sake, later feature engineering will only use the unix `'TIMESTAMP'.`
  From line 38ff it reads line for line of the input file (`test_decoded.txt`) and searches for the appropriate JASON token and then uses a regular expression to extract either the next number or string. While this approach is not the fastest it guaranties that the correct field is extracted, since it has to decode multiple message types(1,3,18,27) and certain message types can include varying number of fields. Further if a field was missing, values where replaced by the default value specified in: http://catb.org/gpsd/AIVDM.html

Decoding and preparing the static set:

- `cl_strip_static.py`
  Similar to the `cl_strip.py` it again creates a file of !AIVDM messages line after line. Since it runs for quite some time it will output every 100000 lines a progress report.
  Further it appends `',0'` to every AIVDM message and then calculates the checksum (line 22ff) and ads it in the appropriate format. Output is written to the file `static_stripped.csv`
- `cl_run_decode_static.sh`
  This script feeds a modified ais.py decoder named ais_static.py. The only changes are that the field length ranges were increased in line 910 ff. (The original values where commented out and can be seen in the previous lines.)
- `cl_create_staticset.py`
  Reads the output of `cl_run_decode_static.sh` and extracts the following fields: `'TIMESTAMP', 'TIME', 'MESSAGE_ID', 'MMSI', 'REPEAT', 'AIS_VERSION', 'IMO_ID', 'CALLSIGN', 'SHIPNAME', 'SHIPTYPE', 'TO_BOW', 'TO_STERN', 'TO_PORT', 'TO_STARBORD', 'EPFD', 'ETA_MONTH', 'ETA_DAY', 'ETA_HOUR', 'ETA_MINUTE', 'DRAUGHT', 'DESTINATION', 'DTE', 'PARTNO', 'VENDORID', 'RAW_MESSAGE'.`

  The script first (line 58ff) checks if it can find the string 'AIVDM', this would mean an error has occurred during decoding and the line either includes the malformed AIVDM message or the decoding error including the AIVDM message. If this happens all fields are left empty, this occurred only about a dozed times.
  The messages in the static data set belong either to type 5 or type 24.
  The script starts with extracting common fields of both message types. Then depending on the message type, fields that are not available to that specific message type are left empty. Message type 24 in addition, is split in two parts

both parts are processed independently and not merged. The way the algorithm works (i.e looking for the field token first) makes particular sense in part 2 of the message type 24 since it can include varying number and types of fields.
Output is written to `static_data.csv`

- cl_create_shipset.py
  Creates ship specific data out of the `static_data.csv` and following fields: `'MESSAGE_ID', 'MMSI', 'CALLSIGN', 'SHIPNAME', 'SHIPTYPE','TO_BOW','TO_STERN','TO_PORT','TO_STARBORD','EPFD','DRAUGHT'`
  This is done in line 46 by grouping on `'MMSI'`, then the command `.first()` will extract the first non empty value of the particular data field (This automatically stitches the type 24 messages together). This could be improved by using the most common non-empty value; if it turns out that future transmission have a lot of transmission errors.

Model and Prediction:

- cl_prediction.py
  This file contains the XGBoost model and all feature engineering and the prediction process. Feature engineering starts at line 60 by extracting additional features from the timestamp. In line 100 we begin to import the ship specific data and calculate the new feature `'SIZE'` which is `'TO_BOW' + 'TO_STERN'`. In line 110 this information is left merged on the train and test data. Following categorical fields `'NAV_STATUS', 'POS_ACCURACY', 'MESSAGE_ID', 'SHIPTYPE'` are one hot encoded in line 125.
  Line 135ff defines the parameters of the tree booster (more information on their meanings can be found here):
  https://xgboost.readthedocs.io/en/latest//parameter.html.
  My code will print the features used (line 183) to the console. The value score in line 199 is used to give the output file a version number. Originally it printed the validating score of the model by using a split train set. This code was kept in the file and is commented out with `##`(double hashes).

Advantages and disadvantages of the approach chosen

Advantages and disadvantages where discussed previously where appropriate.
In summary the XGBoost library seems to be the standard for tree based approaches. The decoding itself is slow in it's current form but rather robust. The generated intermediary files produce additional i/o request, however they can be easily split for parallelization or if the memory size is an issue.

Special guidance given to algorithm based on training

Only the choice of XGBoost hyperparameters
(https://xgboost.readthedocs.io/en/latest//parameter.html)
as previously mentioned is based on previous training using LabelShuffleSplit on the training data. I left the code used as a comment in the program.

Potential improvements to your algorithm

- **Fine Tuning:** The hyper parameters of the model were fixed early, before the static set information was included and only test around previously working parameters in a heuristic manner (=one after the other). A more extensive grid search will definitely find better parameters.
- **Ensemble:** Since this is a very sleek and fast model (**6 min** on the provided Linux server) and due to the XGBoost library a model that also scales favorably on multicore machines, it is good starting point for ensemble learning.
- **Additional Features:** Further if I had the time, I would have added Wave Height and Bathymetry Data.
- **Performance:** Last but not least, the decoding and feature extraction part is definitely the slowest (**81 minutes** on the provided Linux server). I see great potential for performance optimization/ parallelization in this part.

Cheers Christoph Langer