Documentation of the Winning Solution from "TheQuants" Team - Avito Duplicate Ads Detection

AVITO DUPLICATE ADS DETECTION

TheQuants Solution

Peter | Sonny | Marios | Mikel

Table of Contents

1. Summary

In the 'Avito Duplicate Ads Detection' classification challenge sponsored by Avito, we were given approximately 4 million pairs of ads, and tasked with developing a model that can automatically spot which such pairs are duplicates. With more accurate duplicate ad detection, Avito envisages to make it much easier for buyers to find and make their next purchase with an honest seller.... Our approach to this competition was divided into several parts:

1. Generate features (on cleaned or raw data) that would potentially capture the similarity between the contents of 2 ads and could be further divided to more categories (like text similarities or image similarities).
2. Build a number of different classifiers and repressors with a hold out sample.
3. Ensemble the results through weighted rank average of a 2-layer meta-model network (StackNet) .

2. The Team

(From left to right):

Name: Mikel Bober-Irizar
Location: Guildford, United Kingdom
Email: mikel@mxbi.net

Name: Sonny Laskar
Location: India
Email: sonnylaskar@gmail.com

Name: Peter Borrmann
Location: Germany
Email: peter.borrmann@gmail.com

Name: Marios Michailidis
Location: London, United Kingdom
Email: mimarios1@hotmail.com

3. Features Selection / Extraction

Data Cleaning

In order to clean the text, we applied stemming using the NLTK Snowball Stemmer, and removed stopwords/punctuation as well as transforming to lowercase. In some situations we also removed non-alphanumeric characters too.

Feature Engineering Vol 1: Features we actually used

In order to pre-emptively find over-fitting features, we built a script that looks at the changes in the properties (histograms and split purity) of a feature over time, which allowed us to quickly (200ms/feature) identify overfitting features without having to run overnight xgboost jobs.

After removing overfitting features, our final feature space had over 600 features derived from different themes:

General:

- CategoryID, parentCategoryID raw CategoryID, parentCategoryID one-hot (except overfitting ones).
- Price difference / mean.
- Generation3probability (output from model trained to detect generationmethod=3).

Location:

- LocationID & RegionID raw.
- Total latitude/longtitude.
- SameMetro, samelocation, same region etc.
- Distance from city centres (kalingrad, moscow, petersburg, krasnodar, makhachkala, murmansk, perm, omsk, khabarovsk, kluichi, norilsk) Gaussian noise was added to the location features to prevent overfitting to specific locations, whilst allowing xgboost to create its own regions.

All Text:

- Length / difference in length.
- nGrams Features (n = 1,2,3) for title and description (Both Words and Characters).
    - Count of Ngrams (#, Sum, Diff, Max, Min).
    - Length / difference in length.
    - Count of Unique Ngrams.
    - Ratio of Intersect Ngrams.
    - Ratio of Unique Intersect Ngrams.
- Distance Features between the titles and descriptions:
    - Jaccard
    - Cosine
    - Levenshtein
    - Hamming
- Special Character Counting & Ratio Features:
    - Counting & Ratio features of Capital Letters in title and description.
    - Counting & Ratio features of Special Letters (digits, punctuations, etc.) in title and description.
- Similarity between sets of words/characters.
- Fuzzywuzzy distances.
- jellyfish distances.
- Number of overlapping sets of n words (n=1,2,3).
- Matching moving windows of strings.
- Cross-matching columns (eg. title1 with description2).

Bag of words:

For each of the text columns, we created a bag of words for both the intersection of words and the difference in words and encoded these in a sparse format resulting in ~80,000 columns each. We then used this to build Naive Bayes, SGD and similar models to be used as features.

Price Features:

- Price Ratio.
- Is both/one price NaN.
- Total Price.

JSON Features:

- Attribute Counting Features.
    - Sum, diff, max, min.
- Count of Common Attributes Names.
- Count of Common Attributes Values.
- Weights of Evidence on keys/values, XGBoost model on sparse encoded attributes.

Image Features:

- # of Images in each Set.
- Difference Hashing of images.
- Hamming distance between each pair of images.
- Pairwise comparison of file size of each image.
- Pairwise comparison of dimension of each image.
- BRISK keypoint/descriptor matching.
- Image histogram comparisons.
- Dominant colour analysis.
- Uniqueness of images (how many other items have the same images).
- Difference in number of images.

Clusters:

We found clusters of rows by grouping rows which contain the same items (eg. if row1 has items 123, 456 and row2 has items 456, 789 they are in the same cluster). We discovered that the size of these clusters was a very good feature (larger clusters were more likely to be non-duplicates), as well as the fact that clusters always the same generationMethod. Adding cluster-size features gave us a 0.003 to 0.004 improvement.

Feature Engineering Vol 2: The ones that did not make it

Overfitting was probably the biggest problem throughout the competition, and lots of features which (over)performed in validation didnâ€™t do so well on the leaderboard. This is likely because the very powerful features learn to recognise specific products or sellers that do not appear in the test set. Hence, a feature graveyard was necessary evil

TF-IDF:

This was something we tried very early into the competition, adapting our code from the Home Depot competition. Unfortunately, it overfitted very strongly, netting us 0.98 val-auc and only 0.89 on LB. We tried adding noise, reducing complexity, but in the end we gave up.

Word2vec:

We tried both training a model on our cleaned data and using the pretrained model posted in the forums. We tried using word-mover distance from our model as features, but they were rather weak (0.70AUC) so in the end we decided to drop these for simplicity. Using the pre-trained model did not help, as the authors used MyStem for stemming (which is not open-source) so we could not replicate their data cleaning. After doing some transformations on the pre-trained model to try and make it work with our stemming (we got it down to about 20% missing words), it scored the same as our custom word2vec model.

Advanced cluster features:

We tried to expand the gain from our cluster features in several ways. We found that taking the mean prediction for the cluster as well as provided excellent features in validation (50% of gain in xgb importance), however these overfitted. We also tried taking features such as the standard deviation of locations of items in a cluster, but these overfitted too.

Grammar features:

We tried building features to *fingerprint* different types of sellers, such as usage of capital letters, special characters, newlines, punctuation etc. However while these helped a lot in CV, they overfitted on the leaderboard.

Brand violations:

We built some features based around words that could never appear together in duplicate listings. (For example, if one item wrote “iPhone 4s” but the other one wrote “iPhone 5s”, they could not be duplicates). While they worked well at finding non-duplicates, there were just too few cases where these violations occurred to make a difference to the score.

4. Modeling Techniques, CV and Training

Validating

Initially, we were using a random validation set before switching to a set of non-overlapping items, where none of the items in the valset appeared in the train set. This performed somewhat better, however we had failed to notice that the training set was ordered based on time! We later noticed this (inspired by this post) and switched to using last 33% as a valset.

This set correlated relatively well with the leaderboard until the last week, when we were doing meta-modelling and it fell apart - at a point where it would be too much work to switch to a better set. This hurt us a lot towards the end of the competition.

Modelling Vol 1: The ones that made it

In this section we built various models (classifiers and regressors) on different input data each time (since the modelling process was overlapping with the feature engineering process . All models were training with the first 66% of the training data and validated on the remaining 33%. All predictions were saved (so that they can be used later for meta modelling. The most dominant models were:

XGBoost:

Trained with all 587 of our final features with 1000 estimators, maximum depth equal to 20 and minimum child of 10, and particularly high Eta (0.1) - bagged 5 times. We also replaced *nan* values with -1 and *Infinity* values with 99999.99. It scored 0.95143 on private leaderboard. Bagging added 0.00030 approximately.

Keras:

Trained with all our final features, transformed with standard scaler as well as with logarithm plus 1, where all negative features have been replaced with zero. The main architecture involved 3 hidden layers with 800 hidden units plus 60% dropout. The main activation function was Softmax and all intermediate ones were standard rectifiers (ReLU). We bagged it 10 times. It scored 0.94912 on private leaderboard. It gave +0.00080-90 when rank-averaged with the XGBoost model.

Modelling Vol 2 : The ones that didn't

We build a couple of deeper Xgboost models with higher Eta (0.2) that although performed well in cv, they overfitted the leaderboard

We used a couple of models to predict generation method in order to use that as feature for meta-modelling but it did not add anything so we removed it

Meta - Modelling

The previous modelling process generated 14 different models including linear models as well as XGBoosts and NNs , that were later used for meta modelling

For validation purposes we splitted the remaining (33%) data again into 66-33 in order to tune the hyper parameters of our meta-models that used as input the aforementioned 14 models. Sklearn's Random Forest which performed slightly better than XGBoost (0.95290 vs 0.95286). Their rank average yielded our best Leaderboard score of 0.95294

5. Code Description

Below is the description of the codes:

Config.cfg

This file contains many settings like:

- *preprocessing_nthreads* – How many cores to run preprocessing
- *models_nthreads* – How many cores to use during model building process
- *BASE_DIR* – The Base Directory where this file resides
- *cache_loc* – The Cache folder (Used for all Intermediate files)
- Other Params like *train_ItemInfo, train_ItemPairs, test_ItemInfo, test_ItemPairs, category_csv, location_csv, images_root*

Note: When running code from outside the directory where config.cfg resides, all file paths should be absolute. If running the script from inside the config.cfg directory, paths can be relative.

The subfolder layout of images_root does not have to be the same as the original data, as our solution dynamically maps the layout of this folder.

Features Scripts:

- 1_data_preprocessing.py
  This script reads in input data, runs stemming/cleaning on the textual data and then merges the data together based on the specified itemID pairs.

- 2_image_info.py
  This script generates metadata such as size & dimensions from images, as well as generating difference hashes and counting the occurrences of these hashes.

- 3_feature_set1a_ngram.R
  This script generates all NGrams based features (n = 1, 2, 3)

- 3_feature_set1b_nchar.R
  This script generates all NChars based features (n = 1, 2, 3)

- 3_feature_set1c_misc.R
  This script generates many other features of title, description, price, location, images, etc.

- 3_feature_set1d_interaction.R
  This script generates interaction features between "isMetroIdSame", "isLocationIDSame", "isRegionIDSame", "isLongitudeSame", "isLatitudeSame", "isTitleSame", "isdescriptionSame"

- 3_feature_set1e_attribute.R
  This script generates JsonAttribute based features

- 3_feature_set1f_SpecialCounting.R
  This script generates features based on special characters like "digit", "graph", "punct", "xdigit"

- 3_feature_set1g_capitalLetters.R
  This script generates features related to Capital Letters

- 3_feature_set1h_images.R
  This script generates all Image hash features

- 3_feature_set1i_imagesSize.R
  This script generates features using size of each image

- 3_feature_set2a_lev_loc.py:
  This script generates text levenshtein features from text as well as fuzzy location features

- 3_feature_set2b_brisk.py
  This script generates BRISK keypoint/descriptor pairs for the images and then uses these to generate similarity features

- 3_feature_set2c_hist.py
  This script generates image features based on the histograms of images as well as analyzing dominant colours.

- 3_feature_set3a_description.py
  This script generates some features from clean descriptions

- 3_feature_set3b_title.py
  This script generates some features from clean titles

- 3_feature_set3c_json.py
  This script generates some features from clean json

- 3_feature_set3d_json1.py
  This script generates json jaccard similarity

- 3_feature_set3f_hamming.py

This script generates features from image differenceHashes

- 3_feature_set3z_consolidate_internal.R
  This script is used internally by 3_feature_set3z_consolidate.R

- 3_feature_set3z_consolidate.R
  This script consolidates all set3 features and adds few more features.

- 3_feature_set4a_fuzzy.py
  This script creates difference distance metrics between Title, Description and Attributes using the fuzzywuzzy python package

- 3_feature_set4b_fuzzy_clean.py
  This script creates difference distance metrics between Title, Description and Attributes using the fuzzywuzzy python package after applying stemming and cleaning the data.

- 3_feature_set4c_alternate.py
  This script creates similarity features (like count of common words) between different fields of the 2 itemids. For example Title of id1 and description of id2.

- 3_feature_set4d_similarity_clean.py
  This script creates similarity features (like count of common words) between Title, Description and Attributes of the 2 itemIDs.

Consolidation Scripts:

a. 5_consolidate_features.R
   This script consolidates all features into a single feather file.

b. 5_data_postprocessing.py
   This script replaces NaN and Inf values in the final files created by script 3_feature_set3z_consolidate.R

Model Scripts:

a) marios_nnnew_v2.py
b) marios_nnnew_v3.py
c) marios_nnnew_v4.py
d) marios_ridge_v2.py
e) marios_sgd_v2.py
f) marios_xg_v1.py
g) marios_xgrank_v2.py
h) marios_xgrank_v3.py
i) marios_xgregv3.py
j) marios_xgson_v2.py
k) marios_xgson_v3.py
l) marios_xgsonv2_v5.py
m) marios_logit_v2.py
n) marios_nn_v1.py

All these model follow the same structure. A function that reads the data into numpy arrays, separating the data to 66% training and 33% validation. Training models via bagging and saving output predictions for the validation data inside the train folder and printing the validation AUC. Then again retraining using 100% of the training data and scoring the test data via bagging, saving the results into the test folder and generating submission file based on the competition's format

The final model script is **meta_rf_v1.py** .This script performs meta modelling (stacking) and has exactly the same format with the rest with the only difference that the input data is replaced by a function that loops through all the previously saved models and appends ("stacks") them to 1 numpy array before doing the modeling.

All Run Script:

RunAll.sh:
This scripts runs all scripts one after another to generate the final solution

6. Dependencies

- **OS:** Any Linux Distribution (Ubuntu 14.04 Preferred)
- **RAM:** 128GB+ (64GB for feature extraction)
- **CPU:** 36 cores+ (Preferred)
- **GPU:** CUDA-compatible NVIDIA GPU with Compute Capability 3.5+ (TITAN X Preferred)
- **Storage:** 64GB+ (not including input data) - Images on SSD _highly recommended_
- R Version: 3.1+
    - R Packages: data.table, dplyr, dummies, feather, Hmisc, igraph, jsonlite, parallel, raster, readr, reshape2, stringdist, stringr, stylo, textreuse, tidyr, tm, xgboost
- Python Version: 3.5.1
    - Python Packages: Scikit-learn, numpy, pandas, python-Levenshtein, codecs, OpenCV, feather-format, jellyfish, nltk, PIL, fuzzywuzzy, stop_words, haversine
- Python Version: 2.7.1
    - Python Packages: XGBoost (0.4.0), Keras (0.3.2), Theano (0.8.0rc1), scikit-learn, feather-format

All feature generation is done in R and Python3, while all models are run in Python2.

7. How to Generate the Solution (aka README file)

1) Update *config.cfg* and set all config parameters
2) Ensure all directories mentioned in *config.cfg* are 'write-able'
3) Execute *RunAll.sh*

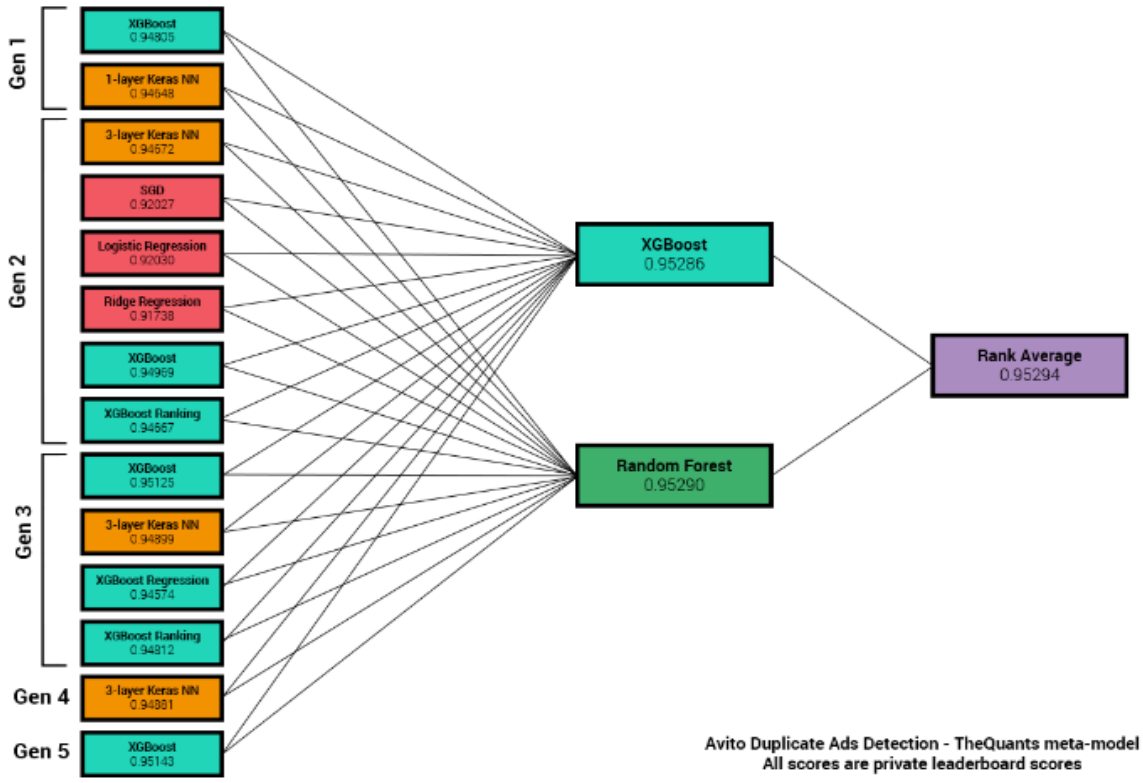
Note: To generate the final submission, it might take several weeks and needs at least 128 GB of RAM.

8. Additional Comments and Observations

It seems that this competition was primarily driven by feature engineering and lots of work in this space could yield good results even with simpler models.

9. Figures

The Modelling and Meta-Modelling process is also illustrated below:



10. References

- *Stacked generalization, Wolpert*
- *Stacked generalization: When does it work?, Ting*
- *Kaggle Ensembling Guide, MLWave*
- *Beat the Benchmark, David Shinn*
- *Stacking code for the BioResponse Kaggle competiton, Olliveti & Solarzano*
- *Extremely Randomized Trees, Geurts et al.*
- *Random Forests, Breiman*
- *Keras: Theano-based Deep Learning library*
- *XGB, Bing xu et al.*
- *Keras*
- *The strength of weak learnability - Robert Schapire*
- *LIBLINEAR -- A Library for Large Linear Classification*
- *Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.*