Avito

Predict deal probability for an online advertisement

by: Itai Yifrach, Stephan Goldberg, Ilai Fallach, Matan Lieber

Table of Contents

Table of Contents **0**
Background **1**
The Problem **1**
Evaluation **1**
Dataset Description **2**
Dataset Analysis **2**
Data Understanding 2
Our Solution **3**
Feature Extraction 3
Text 3
Image 4
Models **4**
LGBM - Decision tree boosting 4
Neural Networks 6
Data vectorization/encoding towards feeding it into the neural network 6
Data vectorization/encoding - Text features 7
Data vectorization/encoding - Categorical features 9
Data vectorization/encoding - Numeric features 9
Neural Networks' Architecture 9
A General Ensemble 11
Conclusion and Results 11
What we did not do (and saw that other gold medalists did)? 11
Insights and Applications **12**
References **12**

Background

Avito.ru is a Russian advertisement website with sections devoted to general good for sale, jobs, real estate, personals, cars for sale, and services.

When selling used goods online, a combination of tiny, nuanced details in a product description can make a big difference in drumming up interest.
Even with an optimized product listing, demand for a product may simply not exist – frustrating sellers who may have over-invested in marketing.

Avito, Russia's largest classified advertisements website, is deeply familiar with this problem. Sellers on their platform sometimes feel frustrated with both too little demand (indicating something is wrong with the product or the product listing) or too much demand (indicating a hot item with a good description was underpriced).

The Problem

Avito Demand Prediction Competition link

Predict deal probability for an online advertisement based on its full description (title, description, images and other ad metadata), its context (geographically where it was posted, similar ads already posted) and historical demand for similar ads in similar contexts.

Evaluation

Root Mean Squared Error (RMSE) for the predicted probabilities (regression problem), as defined by Avito in the competition.

Dataset Description

- **train.csv** - Train data (1503424 rows)
    - price - Ad price.
    - item_seq_number - Ad sequential number for user.
    - activation_date- Date ad was placed.
    - user_type - User type.
    - image - Id code of image. Ties to a jpg file in train_jpg. Not every ad has an image.
    - image_top_1 - Avito's classification code for the image.
    - **deal_probability** - The target variable. This is the likelihood that an ad actually sold something. It's not possible to verify every transaction with certainty, so this column's value can be any float from zero to one.
    - item_id - Ad id.
    - user_id - User id.
    - region - Ad region.
    - city - Ad city.
    - parent_category_name - Top level ad category as classified by Avito's ad model.
    - category_name - Fine grain ad category as classified by Avito's ad model.
    - param_1 - Optional parameter from Avito's ad model.
    - param_2 - Optional parameter from Avito's ad model.
    - param_3 - Optional parameter from Avito's ad model.
    - title - Ad title.
    - description - Ad description.
- **test.csv** - Test data (508438 rows). Same schema as the train data, minus deal_probability.
- **train_jpg.zip** - Images from the ads in train.csv.
- **test_jpg.zip** - Images from the ads in test.csv.

Dataset Analysis

Data Understanding

We found the most significant correlations in the dataset. We visualized the distribution of several features which helped us to decide on feature transformations (e.g. log transformation for the price) and it also helped us to come up with some additional features (weak/power user, count features).

Obviously, we also stayed updated with the most extensive EDA kernels on the competition's discussion page (see under references). We made much more exploration and understanding but due to the space limitations - see *./data_exploration.ipynb* notebook.

Our Solution

Feature Extraction

The feature engineering was composed of 4 main parts: Data Imputation provided features, Text Feature Engineering, Image Feature Engineering and "Others".
The first three has their own notebooks while "Others" are features created in various places, mainly in *./feature_engineering/feature_enrichment.py*. Here we highlight the text and image features. During the process of feature engineering we assessed the importance of created features on our LGBM model to understand where to expand our efforts/invest more time.

Text

As a first step we created **"count"** features for the texts of each ad (e.g word count, capital letter count etc…). Those feature happened to be very informative and critical for our models. Afterwards we tried to examine the "parts of speech" in each text feature (the use of verbs, nouns and adjectives) that are used in each ad. For example, counting the number of adjectives in the title or listing the first three nouns (stemmed) that appear in the description. We invested a lot of time in this analysis and engineering and later discovered that it was **almost useless** (those features did not improve any of our models in a dramatic way).
The last stage was creating a vectorized/numerical representation of ad's text features "as a whole" to feed it into a decision tree or a neural net. For those purposes we used **TF-IDF** and **CountVectorization**. Unsurprisingly, those features were the most significant text features we created. See notebooks for details.

Image

One of the major problem was the amount and size of the data - we had to process 1.5mil images with size of 60GB. To gain the needed computation power, we worked on Intel DevCloud.

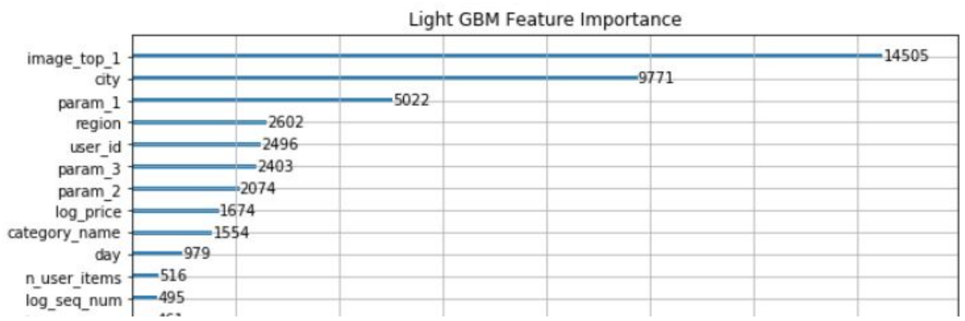
The quality of the advertisement image significantly affects the demand volume of an item. We implemented some ideas which can be used to create new features related to images. These features are indicators for the image quality: Image Size, Image Sharpness, Image Luminance, Image Colorfulness, Image Average Color, Image Dominant Color, Image Keypoints.
We also tried to assess how “descriptive” each image is. For that purpose we used object classification by pre-trained deep-learning models. Our assumption is that the image classification accuracy score will reflect how clear it is for a human to identify it, and effect the chance of buying it. We used pre-trained deep learning models over Keras - ResNet50, InceptionV3 and Xception. From each photo, we predicted the most probable label (for each model) and averaged their probability to a feature named: “Image Confidence”. More details and code can be found in the *./feature_engineering/image_feature_engineering.ipynb* notebook.

Models

LGBM - Decision tree boosting

We’ve decided to begin with a Decision Tree ensembling variation as a starter model. This decision was mainly based on the simplicity of using those kind of models (minimal preprocessing of data is required) and the fact that those models are very descriptive in terms of the importance they give to certain features.
There are several popular algorithms in this family, while ensembling methods rule with superior performance and speed, between three popular decisions: XGBoost, CatBoost and LGBM. After experimenting on initial hyperparameters, we finally chose LGBM as our DT ensembling algorithm (CatBoost showed lower performance and XGboost took much more preprocessing efforts with same potential as LGBM).

Here’s an example of an auxiliary output of the LGBM model:



Outputs as this helped us better understand the significance of each feature we were working with and focusing on the most important features by better pre-process them (better complete missing values, normalize them etc...). For example after witnessing on multiple runs with different hyperparameters the importance of “image_top_1” and “price” we’ve decided to apply deep-learning to complete missing values for those in our train and test sets, i.e writing a neural network classifier that predicts “image_top_1” as a label for records with missing “image_top_1”.

(Please refer to the readme (./README.md) for more reference and actual models. For better understanding this data imputation we recommend reading those notebooks after reading the part about neural networks models below).

Descriptive outputs as above also allowed us picking up the most important features and focus on them while properly feeding the data into our deep learning models.

Using LGBM we’ve arrived to a model with validation error of 0.219 and 0.2244/0.2281 on site’s test data set (public test data / private test data accordingly).

We accomplished it by experimenting with different feature selection and different model parameters. We tried different groups of features and eliminated those who weren’t important (like the part of speech text features we’ve created) or reduced our validation score (features that made the model overfit like ‘item_id’ / ‘day’).

Our best model parameters were achieved by gradually decreasing the learning rate (from 0.1 to 0.03) and increasing the number of leaves (from 32 to 512). The final subset of features we used can be found in the final LGBM model notebook.

We quickly understood (from experimenting with the LGBM and other decision-tree-boosting models) that without neural networks we will not cross the 0.224 test error barrier. Thus we decided to aim for an ensemble model that will combine the predictions of the best two variations of our LGBM model from above (the best two sets of hyperparameters), and a few neural networks that will differ from each other by their architecture and the way they process text features.

Neural Networks

Our exploration of the neural networks models within this project consisted of two phases: the preprocessing of the data towards feeding it into the network (vectorization and encoding) and picking the right architectures of the networks. Here we describe how we've arrived to the networks that our ensemble is consisted of and point out the main conclusions that we've derived during this process.

Data vectorization/encoding towards feeding it into the neural network

This explanation follows the "NN-Model-design-explanation" notebook.
After loading all features we've engineered from the previous part, we choose the ones we'll learn from in this model. This choice is made under the "Vectorize features towards input to an NN" part and based on features' significance as was determined by the LGBM models.

We divide the features into three: text, categorical and numeric.

Obviously, each type gets its own pre processing.

The first task is to vectorize our textual and categorical features that we can feed it as an input for a deep learning model.

Data vectorization/encoding - Text features

All our NN models vary mainly in the way they pre-process the text features which is the most important feature. We explored three ways to represent the texts of the ads we learned from:

1. *Binary Count Vectorization* - For each n-gram that appears in the text feature put a "1", with no importance for the "frequency" of the word in the text. I.e both a word that appears once and a word that appears 17 time are represented as "1"s in the vector that represents the text.

   a. We chose to work with binary representation after experimenting with regular “CountVectorizer”. It’s more problematic to normalize its values/features and working with non-binary vectorization resulted in worse validation scores.

2. *TF-IDF Vectorization* - Vectorize text by TF-IDF (frequency–inverse document frequency). It’s a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. For details see wikipedia page.
   a. As can be seen from the relevant notebooks, the TF-IDF vectorization gave much better validation scores than the *Binary Count Vectorization,* but still we kept the binary CountVec models for maintaining high variance within the ensemble’s models.
3. *Using pre-trained word embeddings and feeding them into an LSTM*
   a. We downloaded and used FastText Russian word embeddings (Facebook’s word embeddings learnt by skipgram on Russian Wikipedia).
   b. We encoded each text as a series of word embeddings and fed those sequences to an LSTM. We use the LSTM output as the text representation (Obviously we include the LSTM in the network and alter its weights while learning to better represent the text representation).
   c. Those networks gave the best validation scores, but took much longer time (and much more resources - 3 days of monster machine on GCP) to train.

For each one of the first two methods we built four models. Two that concatenate all text features (Title, Description, Parameters) and vectorize it (each by its method from those above). And the other two represent each feature as a separate vector.

Each pair contains one model that encode the text by unigrams (“bag of words” encoding) and the other encode bigrams. To sum up those, in total we’ve got 8 models for the first two encoding methods:

	Binary Count Vectorization	**TF-IDF**
Merged texts	Unigrams	Unigrams
	Bigrams	Bigrams

Separated title and description	Unigrams Bigrams	Unigrams Bigrams

We also created four LSTM-based models. Two learn the probabilities solely from the concatenated text features, while the other two combine the first with all the other features. In each pair one model uses Dropout layers within the LSTM (masking out part of the neurons in the recurrent layer) to help generalization and the other does not use Dropout.

Keypoints parameter tuning for text encodings:

1. For all of the above we filter out Russian stopwords (downloaded from from a predefined list.
2. **We do not stem** the words because stemming showed dramatic increase in error rates. We used NLTK's russian stemmer. It just doesn't work well in Russian probably because this process losses too much descriptive information that the original words hold.
3. We tried vectorizing texts (within the TF-IDF and CountVec methods) to various vector sizes. We tried a grid search between 1000 and 200,000. Our vocabulary size is 900,000 (unstemmed, meaning many suffixes for each word). Surprisingly we've got the best result for smaller vector sizes (~7,500).

   For vectors of size > 20,000 the networks overfitted after one epoch and resulted with terrible validation scores.
4. We used a max_df=0.5 filter for both TF-IDF and CountVec methods, meaning a word that appears in more than half of the records is filtered out (kind of a "corpus specific stopword"). It show a nice increase in validation score and learning rate.
5. We do not use min_df as some words are pretty rare but once they appear they are very indicative (e.g some kind of item name\model).
6. We had a problem with LSTM when we stemmed input texts, making the dimensionality of the vocabulary much smaller but losing information that the LSTM and bi-grams models work with (resulting in worse validation score).

You can see all of the above exact implementation in the notebooks for the models that are named accordingly to the way they pre-process text features.

Data vectorization/encoding - Categorical features

We used one hot encodings, first by using label encoder and applying the one hot encoder on the labeled categorical data.

Data vectorization/encoding - Numeric features

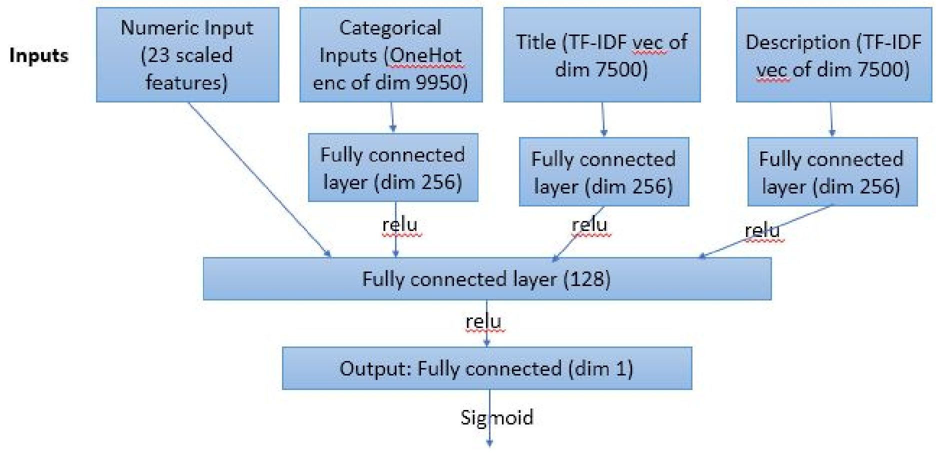
We log scaled each feature with high skewness and/or variance.

After log scaling we clamped the 1% of the top outliers in high-skewed features (such as “price” that had “max int” values).

After scaling and clamping we normalized (L2) each numeric feature in order to allow the network to efficiently process this data (without “exploding gradients”).

Neural Networks’ Architecture

For the TF-IDF and CountVec networks, we had networks of the following structure:


Inputs
Numeric Input (23 scaled features)
Categorical Inputs (OneHot enc of dim 9950)
Title (TF-IDF vec of dim 7500)
Description (TF-IDF vec of dim 7500)
Fully connected layer (dim 256)
Fully connected layer (dim 256)
Fully connected layer (dim 256)
relu
relu
relu
Fully connected layer (128)
relu
Output: Fully connected (dim 1)
Sigmoid

1. The best validation score we managed to achieve with those networks was 0.2281
2. The best test score we managed to achieve with those networks was 0.232 (much worse than the LGBMs)
3. We used L2 regulizers which improved our validation score.
4. We trained the network for only two epochs while enlarging batch size and making verification set smaller. This was done to prevent overfit (which is caused by the textual features).

5. We did not use dropout layers because those did not improve our validation scores.

For LSTM based networks we had similar structure, but instead the textual modules we plugged in an LSTM with additional hidden layer that received FastText word embeddings.

1. This network worked much better, achieving validation score of 0.2272 after just three epochs and test score of 0.2305 (Still as a “single model” it did not beat the LGBM).
2. Training LSTMs in such “combined” networks was a tough task from the computational point of view. We payed for 2 days of a strong machine on GCP (60GB RAM, 2 GPUs, 8 cores) and achieved the scores mentioned above. We wanted to check how further can we go with our model given more resources (time and power on GCP/Other cloud platform). We deleted all components and trained **Just** an LSTM on our data’s text features (no other numeric / categorical inputs). Surprisingly we have achieved 0.229 validation after 3 epochs (which is not much worse). That tells us that given more time and resources with higher computational power we could train our combined network to achieve much better results.

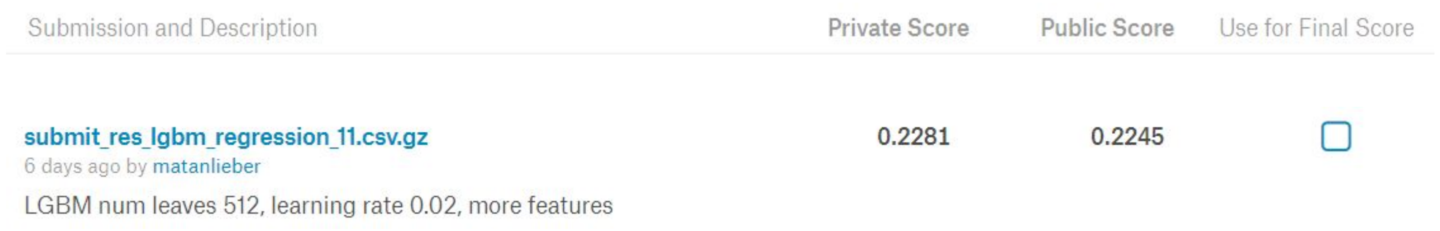
A General Ensemble

After the disappointment that followed the test scores of our neural networks we tried to ensemble them together. It indeed helped to reduce test error, but still did not get close to the LGBM model. After trying ensembling together the Neural nets and the LGBMs, we got 0.2250 public test error (compared to 0.2245 public test error for LGBM only).

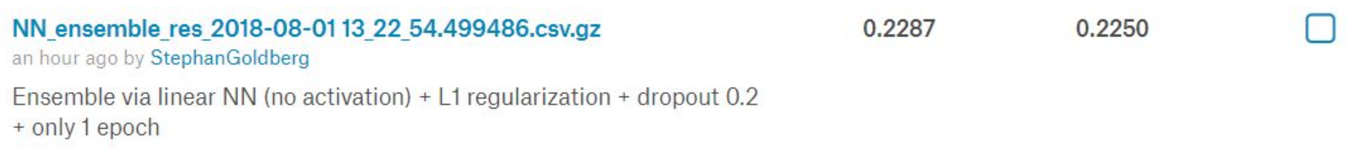
The ensembling techniques we used can be seen in the relevant notebooks.

Conclusion and Results

We ended up with final scores of:



Submission and Description	Private Score	Public Score	Use for Final Score
submit_res_lgbm_regression_11.csv.gz 6 days ago by matanlieber LGBM num leaves 512, learning rate 0.02, more features	0.2281	0.2245	☐


NN_ensemble_res_2018-08-01 13_22_54.499486.csv.gz 0.2287 0.2250
an hour ago by StephanGoldberg
Ensemble via linear NN (no activation) + L1 regularization + dropout 0.2 + only 1 epoch

Which would grant us the 1115th place (out of 1917) in the competition (120 places below our place in the day the competition officially ended). But it worths mentioning that the difference between our best score to the winners is only 1.3% (absolute distance).

The experience was at first frustrating but in retrospective we can only thank for it. It was the most enlightening expertise we had with machine learning in the university and it **underlined the vast differences between machine learning courses in the university and real life data challenges.**

What we did not do (and saw that other gold medalists did)?

1. We did not stack our models one upon another. I.e we did build a meta network that ensembles all our models (TF-IDF, CountVec, LSTM, LGBM and others) and optimizes (backpropagation) the loss upon the model as a whole (we trained separate models and ensemble the results).
2. We did not ensemble an XGboost model due to time/resource/priorities limitation.
3. We did not feed images as raw pixels to a CNN (as one of the stacked ensemble model).
4. We did not train deeper models. Either because it made us overfit, or in other cases it took too long for the model to learn (and as mentioned above, even the Intel cluster we had wasn't enough for it. We had to pay for a GCP machine by ourselves). See gold solutions in the discussions to get a proportion for the depth of their stacks and the machines they ran their algorithms for 24/7 on.

Insights and Applications

According to our models, the most important features are image_top_1 (which is image classification to categories), price and some of the extracted features from the text and images.

We can combine all the information to advice the user, which is applying a new ad, which parameters to improve in order to increase the probability of the ad to be sold. For example, we can tell the user online if his image is quality enough, or if his text is informative/related to his ad.

References

1. Feature engineering:
    a. https://www.kaggle.com/bminixhofer/aggregated-features-lightgbm
2. Main EDAs that we stayed updated with:
    a. https://www.kaggle.com/codename007/avito-eda-fe-time-series-dt-visualization/notebook
    b. https://kaggle.com/kabure/extensive-eda-of-deal-probability/notebook
3. Images:
    a. how to calculate image luminance? https://stackoverflow.com/questions/596216/formula-to-determine-brightness-of-rgb-color
    b. how to calculate image colorfulness? https://www.pyimagesearch.com/2017/06/05/computing-image-colorfulness-with-opencv-and-python/
    c. how to calculate image sharpness? https://stackoverflow.com/questions/28717054/calculating-sharpness-of-an-image
    d. how to calculate average color and dominant color? https://zeevgilovitz.com/detecting-dominant-colours-in-python
    e. how to find image keypoints? https://stackoverflow.com/questions/29133085/what-are-keypoints-in-image-processing
4. Many references regarding our NN models and text feature eng appear as comments in the notebooks.