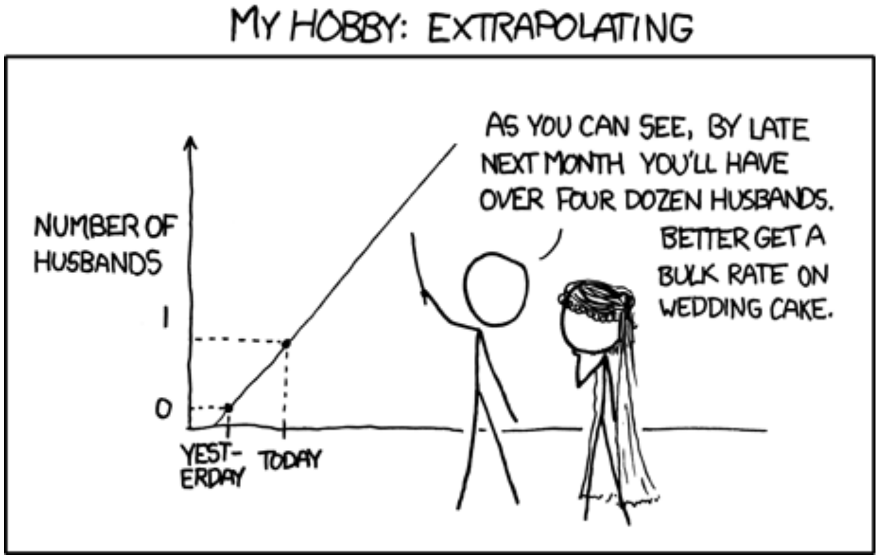
Yury Kashnitskiy Follow
Data Scientist at Mail.Ru Group
Feb 26 · 9 min read

Open Machine Learning Course. Topic 4. Linear Classification and Regression


MY HOBBY: EXTRAPOLATING
NUMBER OF HUSBANDS
1
0
YEST-ERDAY
TODAY
AS YOU CAN SEE, BY LATE NEXT MONTH YOU'LL HAVE OVER FOUR DOZEN HUSBANDS. BETTER GET A BULK RATE ON WEDDING CAKE.

xkcd.com

Welcome to the 4-th week of our course! Now we will present our most important topic—linear models. If you have your data prepared and want to start training models, then you will most probably first try either linear or logistic regression, depending on your task (regression or classification).

This week's material covers both theory of linear models and practical aspects of their usage in real-world tasks. There's going to be a lot of math in this topic, and we won't even try to render all the formulas on Medium. Instead, we provide a Jupyter Notebook for each part of this article. In the assignment, you'll beat two simple benchmarks in a Kaggle competition solving a problem of identifying a user based on her session of visited websites.

Article outline

Part1. Regression, nbviewer

Part 6. Kaggle Inclass competition "Catch Me If You Can"

Competition's page

We will be solving the intruder detection problem analyzing users' behavior on the Internet. It is a complicated and interesting problem combining data analysis and behavioral psychology. As an illustration of one of such tasks, Yandex solves the mailbox intruder detection problem based on the user's behavior patterns. In a nutshell, intruder's behavior patterns may differ from those of the mailbox owner:

- the breaker may not delete emails right after they are read as the mailbox owner may do;
- the intruder may mark emails and even move the cursor differently;
- etc.

So the intruder could be detected and thrown out from the mailbox forcing the user to authenticate via SMS-code.

Similar approaches are being developed in Google Analytics and described in scientific research papers. You can find more on this topic by searching "Traversal Pattern Mining" and "Sequential Pattern Mining".

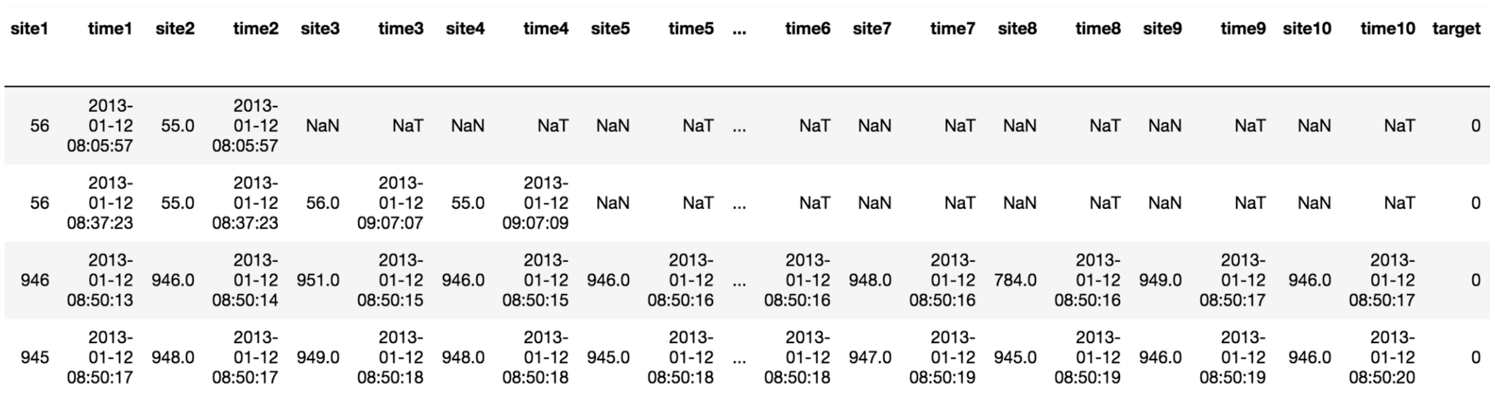
In this competition we are going to solve a similar problem: our algorithm is supposed to analyze the sequence of websites consequently visited by a particular person and predict whether this person is a user named Alice or an intruder (somebody else). As a metric, we will use ROC AUC.

Data Downloading and Transformation

Register on Kaggle, if you have not done it before. Go to the competition page and download the data.

First, load the training and test sets. Then explore the data and perform a couple of simple exercises:

```
%matplotlib inline
from matplotlib import pyplot as plt
import seaborn as sns
import pickle
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
from scipy.sparse import hstack
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
from sklearn.linear_model import LogisticRegression
# Load the training and test data sets
train_df = pd.read_csv('../../data/websites_train_sess
                       index_col='session_id')
test_df = pd.read_csv('../../data/websites_test_sessic
                      index_col='session_id')
```



site1	time1	site2	time2	site3	time3	site4	time4	site5	time5	...	time6	site7	time7	site8	time8	site9	time9	site10	time10	target
56	2013-01-12 08:05:57	55.0	2013-01-12 08:05:57	NaN	NaT	NaN	NaT	NaN	NaT	...	NaT	NaN	NaT	NaN	NaT	NaN	NaT	NaN	NaT	0
56	2013-01-12 08:37:23	55.0	2013-01-12 08:37:23	56.0	2013-01-12 09:07:07	55.0	2013-01-12 09:07:09	NaN	NaT	...	NaT	NaN	NaT	NaN	NaT	NaN	NaT	NaN	NaT	0
946	2013-01-12 08:50:13	946.0	2013-01-12 08:50:14	951.0	2013-01-12 08:50:15	946.0	2013-01-12 08:50:15	946.0	2013-01-12 08:50:16	...	2013-01-12 08:50:16	948.0	2013-01-12 08:50:16	784.0	2013-01-12 08:50:16	949.0	2013-01-12 08:50:17	946.0	2013-01-12 08:50:17	0
945	2013-01-12 08:50:17	948.0	2013-01-12 08:50:17	949.0	2013-01-12 08:50:18	948.0	2013-01-12 08:50:18	945.0	2013-01-12 08:50:18	...	2013-01-12 08:50:18	947.0	2013-01-12 08:50:19	945.0	2013-01-12 08:50:19	946.0	2013-01-12 08:50:19	946.0	2013-01-12 08:50:20	0

The training dataset contains the following features:

- **site1**—id of the first visited website in the session;
- **time1**—visiting time for the first website in the session;
- …
- **site10**—id of the tenth visited website in the session;
- **time10**—visiting time for the tenth website in the session;
- **target**—target variable, takes the value of 1 for Alice's sessions, and 0 for the other users' sessions.

User sessions are chosen in such a way that they are not longer than half an hour and/or contain more than ten websites; i.e. a session is considered as ended either if the user has visited ten websites or if the session has lasted over thirty minutes.

There are some empty values in the table, which means that some sessions contain less than ten websites. Replace empty values with 0 and change column types to integer. Also, load the website dictionary and see how it looks:

```
# Change site1, ..., site10 columns type to integer an
sites = ['site%s' % i for i in range(1, 11)]
train_df[sites] = train_df[sites].fillna(0).astype('in
test_df[sites] = test_df[sites].fillna(0).astype('int'
# Load website dictionary
with open(r"../../data/site_dic.pkl", "rb") as input_f
    site_dict = pickle.load(input_file)
# Create dataframe for the dictionary
sites_dict = pd.DataFrame(list(site_dict.keys())
```

```
# Websites total: 48371
```

	site
25075	www.abmecatronique.com
13997	groups.live.com
42436	majeureliguefootball.wordpress.com
30911	cdt46.media.tourinsoft.eu
8104	www.hdwallpapers.eu

In order to train our first model, we need to prepare the data. First of all, exclude the target variable from the training set. Now both training and test sets have the same number of columns, and we can aggregate them into one dataframe. Thus, all transformations will be performed simultaneously on both the training and test datasets.

On the one hand, it leads to the fact that both of our datasets have one feature space (so you don't have to worry that you may have forgotten to transform a feature in one of the datasets). On the other hand, the processing time will increase. In case of enormously large sets, it may turn out that it is impossible to transform both datasets simultaneously (and sometimes you have to split your transformations into several stages, separately for the train/test dataset). In our case, we are going to perform all the transformations for the united dataframe at once; and, before training the model or making predictions, we will just use the corresponding part of it.

```
# Our target variable
y_train = train_df['target']
# United dataframe of the initial data
full_df = pd.concat([train_df.drop('target', axis=1), t
# Index to split the training and test data sets
```

For the sake of simplicity, we will use only the visited websites in the session (and we will not take into account the timestamp features). The point behind this data selection is: *Alice has her favorite sites, and the more often you see these sites in the session, the higher the probability that this is an Alice's session, and vice versa.*

Let's prepare the data. We will keep only the features `site1, site2, … , site10` in the dataframe. Keep in mind that missing values were replaced with zeros. Here is how the first rows of the dataframe look like:

```
# Dataframe with indices of visited websites
full_sites = full_df[sites]
full_sites.head()
```

	site1	site2	site3	site4	site5	site6	site7	site8	site9	site10
session_id										
21669	56	55	0	0	0	0	0	0	0	0
54843	56	55	56	55	0	0	0	0	0	0
77292	946	946	951	946	946	945	948	784	949	946
114021	945	948	949	948	945	946	947	945	946	946
146670	947	950	948	947	950	952	946	951	946	947

Sessions are the sequences of website indices, and such representation of data is inconvenient for linear methods. According to our hypothesis (Alice has favorite websites), we need to transform this dataframe so that each website has the corresponding feature (column) which value is equal to the number of visits on this website within the session. All of this can be done in two lines:

```
# sequence of indices
sites_flatten = full_sites.values.flatten()
# and the matrix we are looking for
full_sites_sparse = csr_matrix(([1] * sites_flatten.sha
                                sites_flatten,
```

If you understand what just happened here, then you can skip the next section (perhaps, you can handle logistic regression too?). If not, then let us figure it out.

Sparse Matrices

Let's estimate how much memory it would require to store our data in the example above. Our united dataframe contains 336 thousand samples of 48 thousand integer features in each. It's easy to calculate the required amount of memory, roughly:

336K * 48K * 8 bytes = 16M * 8 bytes = 128 GB

Obviously, mere mortals don't have such volumes of memory (strictly speaking, Python may allow you to create such a matrix, but it would not be easy to do anything with it). An interesting fact is that most of the elements of our matrix are zeros. If we counted non-zero elements, then it would make out about 1.8 million, i.e. slightly more than 10% of all matrix elements. Such a matrix, where most elements are zeros, is called *sparse*, and the ratio between the number of zero elements and the total number of elements is called the *sparseness of the matrix*.

To work with such matrices, you can use `scipy.sparse` library, check the documentation to understand what possible types of sparse matrices are, how to work with them and in which cases their usage is most effective. You can learn how they are arranged, for example, be reading the Wikipedia article. Note that a sparse matrix contains only non-zero elements. Finally, you can get the allocated memory size (significant memory savings are obvious):

```
# How much memory does a sparse matrix occupy?
print('{0} elements * {1} bytes = {2} bytes'.
      format(full_sites_sparse.count_nonzero(), 8,
             full_sites_sparse.count_nonzero() * 8))
# Or just like this:
print('sparse_matrix_size = {0} bytes'
```

```
1866898 elements * 8 bytes = 14935184 bytes
sparse_matrix_size = 14935184 bytes
```

Let's explore how the matrix with the websites has been formed using a mini example. Suppose we have the following table with user sessions:

id	site1	site2	site3
1	1	0	0
2	1	3	1
3	2	3	4

There are 3 sessions, and no more than 3 websites in each. Users visited four different sites in total (there are numbers from 1 to 4 in the table cells). And let us assume that:

1. vk.com
2. habrahabr.ru
3. yandex.ru
4. ods.ai

If the user has visited less than 3 websites during the session, the last few values will be zero. We want to convert the original dataframe in such a way that each session would have the corresponding row which shows the number of visits on each particular site; i.e. we want to transform the previous table into the following form:

id	vk.com	habrahabr.ru	yandex.ru	ods.ai
1	1	0	0	0
2	2	0	1	0
3	0	1	1	1

To do this, use the constructor: `csr_matrix ((data, indices, indptr))` and create a frequency table (see examples, code and comments on the links above to see how it works). Here, we set all the parameters explicitly for greater clarity:

```
# data, create the list of ones, length of which equal
# in the initial dataframe (9)
# By summing the number of ones in the cell, we get th
# number of visits to a particular site per session
data = [1] * 9
# To do this, you need to correctly distribute the one
# Indices - website ids, i.e. columns of a new matrix.
# We will sum ones up grouping them by sessions (ids)
indices = [1, 0, 0, 1, 3, 1, 2, 3, 4]
# Indices for the division into rows (sessions)
# For example, line 0 is the elements between the indi
# the rightmost value is not included
```

```
matrix([[2, 1, 0, 0, 0],
        [0, 2, 0, 1, 0],
        [0, 0, 1, 1, 1]])
```

As you might have noticed, the number of the columns in the resulting matrix is not four (by the number of different websites), but five. A zero column has been added, which shows on how many sites the session was shorter (in our mini example we took sessions of length 3). This column is excessive and it should be removed from the dataframe.

Another benefit of using sparse matrices is that there are special implementations of both matrix operations and machine learning algorithms for them, which sometimes allows to significantly accelerate operations due to the data structure peculiarities. This applies to logistic regression as well. Now, everything is ready to build our first model.

3. Training the first model

Let’s build our first model, using logistic regression implementation from `Sklearn` with default parameters. We will use the first 90% of the data for training (the training data set is sorted by time), and the remaining 10% for validation. Let's write a simple function that returns the quality of the model, and then train our first classifier:

```
def get_auc_lr_valid(X, y, C=1.0, seed=17, ratio = 0.9
    # Split the data into the training and validation
    idx = int(round(X.shape[0] * ratio))
    # Classifier training
    lr = LogisticRegression(C=C, random_state=seed,
                            solver='lbfgs', n_jobs=-1)
    # Prediction for validation set
    y_pred = lr.predict_proba(X[idx:, :])[:, 1]
    # Calculate the quality
    score = roc_auc_score(y[idx:], y_pred)

    return score
```

```
0.9198622553850315
CPU times: user 138 ms, sys: 77.1 ms, total: 216 ms
Wall time: 2.74 s
```

The first model demonstrated the quality of approximately 0.92 ROC AUC on the validation set. Let’s take it as the first baseline and a starting point. To make a prediction on the test set, *we need to train the model again on the entire training dataset* (until this moment, our model used only part of the data for training), which will increase its generalizing ability:

```
# Function for writing predictions to a file
def write_to_submission_file(predicted_labels, out_fil
                             target='target', index_la
    predicted_df = pd.DataFrame(predicted_labels,
                                index = np.arange(1,
                                                  pred
                                columns=[target])
    predicted_df.to_csv(out_file, index_label=index_la
# Train the model on the whole training data set
# Use random_state=17 for repeatability
# Parameter C=1 by default, but here we set it explici
lr = LogisticRegression(C=1.0, solver='lbfgs',
```

If you follow these steps and upload the answer to the competition page, then you should get the quality of `ROC AUC = 0.91707` on the public leaderboard.

Assignment #4

In the assignment, your task will be to further improve the model through feature engineering, feature scaling, and regularization. Feature engineering is the most interesting part of Data Scientist's work and very often it can boost the performance in your ML task. You'll first try to add some obvious features (hour and day of attending a site, number of sites in a session etc.). We encourage you to try new ideas and models throughout the course, and participate in the competition —it's fun!

You need to fill in the required code in the notebook and pick answers in this Google form.

Hard deadline: March 11, 23:59 CET.

Useful resources

- A nice and concise overview of linear models is given in the book "Deep Learning" (I. Goodfellow, Y. Bengio, and A. Courville).
- Linear models are covered practically in every ML book. We recommend "Pattern Recognition and Machine Learning" (C.

Bishop) and "Machine Learning: A Probabilistic Perspective" (K. Murphy).

- If you prefer a thorough overview of linear model from a statistician's viewpoint, then look at "The elements of statistical learning" (T. Hastie, R. Tibshirani, and J. Friedman).
- The book "Machine Learning in Action" (P. Harrington) will walk you through implementations of classic ML algorithms in pure Python.
- Scikit-learn library. These guys work hard on writing really clear documentation.
- Scipy 2017 scikit-learn tutorial by Alex Gramfort and Andreas Mueller.
- One more ML course with very good materials.
- Implementations of many ML algorithms. Search for linear regression and logistic regression.
- And many others, feel free to share in comments.

· · ·

Author: Yury Kashnitsky, Data Scientist at Mail.Ru Group. Translated and edited by Yuriy Isakov, Christina Butsko, Nerses Bagiyan, Yulia Klimushina, Anastasia Manokhina, Evgeniy Mashkin, Egor Polusmak, and Yuanyuan Pao.