Approaches to the Kaggle Driver Telematics Analysis (team Moist Tentaclii)

Viktor Evstratov, George Oblapenko

Contents

1 Introduction

The aim of the competition is as follows: given a set of 200 files containing trip data (x and y coordinates of a cars position taken at 1-second intervals), find which

of the trips do not belong to the driver (it can be assumed that the majority of the trips correspond to one driver). The total amount of drivers is 2736 (I know that by heart, since I typed things like **for** i **in** range(2736): quite a lot). What follows is a description of the approaches we used to try and solve this task. Since we didn't rank even in the top 10 percent, we give only a basic description of what we did.

2 Features description

We used a large amount of simple velocity-based features (percentiles of velocity, heading changes, acceleration, decceleration, etc.; minimum and maximum values, average values of these quantities), spatial-based features (the distance of the path's center of mass from a straight line connecting the start and end points, the distance of the path's center of mass from the point $(0, 0)$, the maximum distance of the path from a straight line connecting the start and end points, the distance from the start point to the end point, the total path length) and the duration (in seconds) of the trip.

More complex features included:

- $chaoticness$ — a measure of how many loops, self-intersections, etc. the path has (this feature was derived empirically). It is calculated as follows: $chaoticness = std(dr/dt)/mean(dr/dt)$, where r is the distance from the point on the path to the point $(0, 0)$. Here the derivative may assume negative values.
- $local\_maxima$ — a measure of how many times the driver comes closer to the starting point than he previously was. The signed derivative dr/dt is taken, and smoothed with a Hanning window of size 10. Then, the amount of local maxima in the resulting array is calculated.
- $fourier(i)\_rel\_energy, i \in 1, 2, 3, 4, 5, 6$ — a STFT was taken of the radial coordinate, resulting in 6 bands in the spectrum. The mean over time of each band was divided by the maximum value over all bands over time
- $fourier\_avg$ — the average of all bands over time
- $fourier\_std$ — the standard deviation of all bands over time

- $decc\_nonlinearity\_avg$, $acc\_nonlinearity\_avg$ — a smoothed version of the velocity was taken, and local maxima/minima were found. For each period of acceleration/decceleration (a period of time between a local minima and a local maxima or vice versa), we calculated the maximum difference between a linear function connecting the points and the true values of the velocity in the time period (a measure of non-linearity of the acceleration/decceleration), and then calculated the averages of these differences
- crv_spd — a set of features describing average velocities on turns of different curvature. We split all the turns in a trip into different bins and for each bin calculated the average velocity.
- Various features concerning standstill times (not sure what exactly they were, Viktor was the one who extracted them)

2.1 Path matching

We didn't try path matching for too long (which probably was a mistake). Our approach was to first extract a simple approximation of a path (the path's skeleton), this was done recursively:

- Approximate a path with a straight line (connecting the start and end points)
- Find the furthest point from the straight line
- Replace the straight line with two line segments (connecting the start, furthest and end points)
- Repeat the procedure for each resulting line segment, splitting it until the angle between the two new line segments is too obtuse

We tried DTW on these skeletons, but didn't spend much time fine-tuning the parameters and abandoned path matching (since we didn't get any boost from it). However we used 6 features from the skeletons: for each angle in the skeleton path, we found from which of the following intervals it is: $[0, \pi/6]$, $[\pi/6, 2\pi/6]$, . . . , $[5\pi/6, \pi]$, then we found the maximum velocity in a small vicinity of the angle point. Using only these 6 features and a Random Forest Classifier gave a result of 0.87105. Another feature set was the amount of angles for each trip in the intervals given above, using only this feature set gave a score of about 0.83.

3 Classification approaches

For finding trips not belonging to a certain driver, we tried various outlier detection and classification methods. Note that the best result listed for each method is not the highest we could've achieved: we abandoned most approaches quite quickly, and therefore, did not test them on features we obtained later on.

3.1 Survival function, best result: 0.54002

For one feature, the idea is simple: calculate the z-score of the feature, assume the z-score to be normally distributed, and assign to the trip the probability as follows: $p = 1 - CDF(z) = SF(z)$, where z is the z-score of the feature, CDF is the cumulative distribution function of the normal distribution, SF is the survival function of the normal distribution. (Looking at it now, we should've used $SF(|z|)$, but the method in general does not give a high score anyway).

For multiple features, the survival functions of the z-scores were calculated independently. We tried various ways of averaging the results, out of the ones we tried, the highest score was obtained using the following weighting procedure: for each k-th feature, find its maximum max_k and median med_k over the 200 trips. The weight for feature k is calculated using the formula $w_k = w'_k / \sum_k w'_k$, where w'_k are the unnormalized weights: $w'_k = |(max_k - med_k)/med_k|$.

3.2 OneClassSVM, best result: 0.58776

The **scikit-learn** package for Python provides a One-class SVM implementation for outlier detection. The SVM produces a binary output. The best result was obtained by taking a PCA of the normalized feature array (an array of feature z-scores) (unfortunately, I forgot to record which set of features exactly was used) and then taking the top 30 transformed features, taking their z-score and feeding them to the SVM. The method allows to specify which points should have more "attention" paid to them, and we tried that, using the results of the Convex Hull method (see below) to give higher weights to points with lower predicted probability. However, this gave a lower score than without the weighting. A different set of weights might've proved more effective, but at that moment we had far more effective approaches.

3.3 Convex Hull, best result: 0.63091

The idea of this method is as follows: given m features, a driver's dataset corresponds to a set of 200 points in $\mathbb{R}^m$. We take the convex hull of the set, and to all of the points in the hull assign a "weight" of 0. We then remove them from the set, and repeat the procedure, assigning a "weight" of 1 to the next set of points in the convex hull, and so on, until there are no points left. For each trip the probability is assigned as the weight of the corresponding point divided by the maximum weight, so points in the first convex hull have a probability of 0, and so on.

A too high value of m will exhaust the set of points too quickly, while a too low value will not take into account the complexity of the dataset. $m = 3$ was chosen after some tests, and overall, up to 4 convex hulls were calculated, each for a different set of 3 features, and their average was taken. The method can sometimes fail to work (when due to certain values of the coordinates of the points, it's impossible to build a convex hull). For these drivers, the survival function method was used.

On the following set of features: $dist\_total$, $local\_maxima\_rel$, $max\_dist\_from\_straight$, acc_avg, $fourier\_std$, $decc\_med$, the method gave a score of 0.62433. The best result was obtained by taking the PCA transform of a large amount of normalized features, and then taking the first 12 transformed features (again, the exact original feature list is not recorded). Using the PCA-transformed features also solved the problem of the method failing for some drivers – while using ordinary sets of features led to 20-30 errors for the total set of drivers, no errors were observed when using the PCA-transformed features.

3.4 Denoising Autoencoders, best result: 0.49407

The idea of this method is to build a neural network of stacked denoising autoencoders, train it for each driver, feed the net the features for each trip and reconstruct the original input from the output of the net, and measure the error of the reconstructed input. Outliers should have a higher error than non-outliers.

We have little experience with training/designing/working with neural networks, so we did not adjust the parameters a lot. We used two layers, the first with dimension 60, the second with dimension 40 and about 90 original input features. This worked poorly, probably due to a poor choice of parameters, a small train size (only 200 trips), a high amount of features and our inexperience with neural networks in general.

3.5 Classifier-based approaches, best result: 0.88015

Another idea is to take a driver's trip set, temporarily assign a label of 1 to each element, takes trips from some other drivers, and assign a label of 0 to them. Then train a classifier on this set, and predict the probability of the original driver's trips belonging to class 1. Classification methods such as AdaBoost and Gradient Boosting might not perform well under such circumstances, since they are sensitive to outliers in the training set. We will refer to the driver we are calculating the probabilities for as "our" driver and other drivers as "other" drivers.

3.5.1 Basic Logistic Regression, best result: 0.86264

This is the basic method described above, using logistic regression as a classification method. We used trips from 150 other drivers (this gave a boost of about 0.015 compared to using 10 drivers), setting the **class_weight** parameter to **'auto'** (see **LogisticRegression** documentation in the **scikit-learn** package), since there is obviously a large number of training examples with a label of 0. Using a L1 norm for the regularization also gave a slight boost (0.005), however, it increases computation times significantly. We used a regularization constant $C = 0.62$.

3.5.2 Pre-filtered Logistic Regression

This method is nearly equivalent to the above one, except that it uses some existing set of probabilities to filter out a subset of our drivers trips from the train set (trips with a low probability). We used the result of a basic logistic regression as the "filter", and discarded 15 percent of the trips with the lowest probability. We also used a L1 norm and a regularization constant of $C = 0.62$ (since we assumed such an approach may lead to over-fitting, indeed, higher values of C produced lower scores). We tried finding an optimum combination of the amount of other drivers, amount of discarded trips and the constant C by doing about 10 submissions with various values and then fitting a curve to the resulting data, the optimum values produced by such a method were close to those we used in the end, those being: 150 other drivers, 15 percent of trips discarded and $C = 0.62$.

3.5.3 Logistic Regression with Train set filtering

This idea also utilizes discarding of trips, however, we discard trips from other drivers in the train set. We train a logistic regression classifier on our driver and a certain amount of other drivers, predict trip probabilities both for our driver and

the other drivers, and then discard those trips from other drivers on which the classifier gives a low result (i.e. those on which it works well and says that the trip is not from our driver). Then we re-train the classifier on the remainder of the test set. The idea is to force the classifier to learn examples near the "decision boundary" and feed it outlier-labeled trips that are closer to the original driver's ones. The opposite idea (removing the trips on which the classifier gives a high result) resulted in a worse score than using a simple classifier.

We used 300 other drivers and half of the trips (30000 trips) based on how low the classifier assigned the probability to them. Again, we used L1 regularization.

3.5.4 Pre-filtered Logistic Regression with Train set filtering, best result: 0.86586

A combination of the two approaches shown above, we not only pre-filter the train set, we also train the classifier twice, discarding trips from other drivers which the classifier says are different from our driver.

3.5.5 Random Forest Classifier, best result: 0.88015

The best result using a Random Forest Classifier was equal to 0.88015 (we used a large feature set for that).

4 Best submission description, best result: 0.89286

We took our top N submissions and the created a new submission which was a weighted sum of those submissions, with the weights being the Public Leaderboard scores (divided by their sum, of course). Tried out a few variations of this method (changing the value of N, discarding submissions which scored high, but were similar in terms of the classifier and feature set used to higher-scoring submissions), one of them gave a Public Leaderboard score of 0.89286 (our top 10 not-too-similar submissions were used).