Otto Group Product Classification Challenge

Hoang Duong

May 19, 2015

1 Introduction

The Otto Group Product Classification Challenge is the biggest Kaggle competition to date with 3590 participating teams. We (my teammate Nicholas Guttenberg and I) participated under team Optimistically Convergence, which ended up at the 6th position on the Leader Board. Through this competition, I have learned a lot, and felt the urge to make a documentation for others reference, and more importantly my future reference.

Following is the official description of the competition:

> "For this competition, we have provided a dataset with 93 features for more than 200,000 products. The objective is to build a predictive model which is able to distinguish between our main product categories"

The training dataset has 61,878 rows, while the testing dataset has 144,368 rows. It is a multi-class classification task, with 9 classes representing 9 different product categories. The classes are not balance, as depicted in Figure 1. Performance is measured by log-loss. The competition attracts a record number of participants due the high quality data that can easily fits into a personal laptop, but not too small to make it uninteresting. Minimal feature engineering was needed, as all the features were count of some events not disclosed to participants.

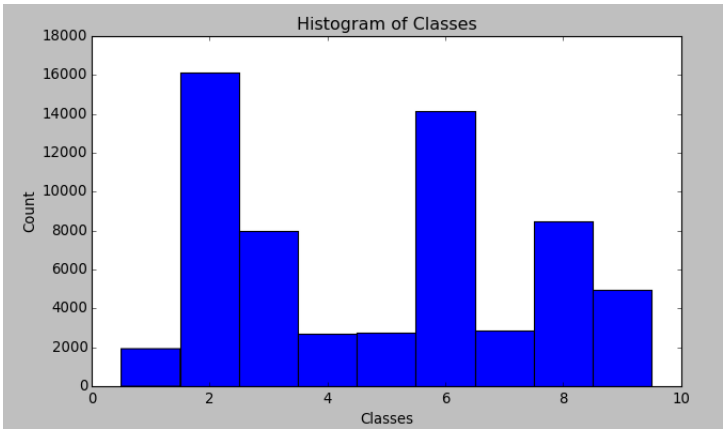

Figure 1: Histogram of Classes in Training Data

My philosophy for this dataset was to keep things as systematic as possible. I tried to avoid manual model tuning, and laborious feature engineering. Paul Duan solution to Amazon Employee Access competition inspire my code style a lot. I've also borrow a lots of ideas from fellow Kagglers through actively reading the forums and past solutions. Finally, I was doing a project on Hyper-parameter tuning alongside with this Kaggle competition, as part of my Convex Optimization class, which proves to be helpful at many points. The main work was done in Python, using extensively scikit-learn library.

2 Feature Engineering

As mentioned, the dataset is already very nice, so minimal feature engineering was needed. I mainly use the following three datasets:

1. The original dataset (80% of the cells are zero, the rest are integers that count events)
2. The $\log(x+1)$ transformed dataset
3. The TF-IDF transformed dataset, as suggested by Kaggle user Abhishek.

Beside this, I did add the square terms for linear models (Logistic Regression, Linear SVM). I also tried using K-Means with different number of centers (ranging from 10, 100, to 200) to create new features by calculating the distance from each point to the centers. This turned out not to help much, mainly because of high dimension issue.

In Figure , we plot the pseudo-correlation of each of the 93 features, with each of the 9 class. The pseudo-correlation between a feature and a class is just $2\left(|\text{auc}(feat_i, class_j)| - 0.5\right)$, for auc is the area under curve of the ROC curve, when using feature i^{th} as a predictor for class j^{th}. We note that a few class are much easier to classify than the other, for example Class 5 (denoted as 4 in the figure for index starting from 0).

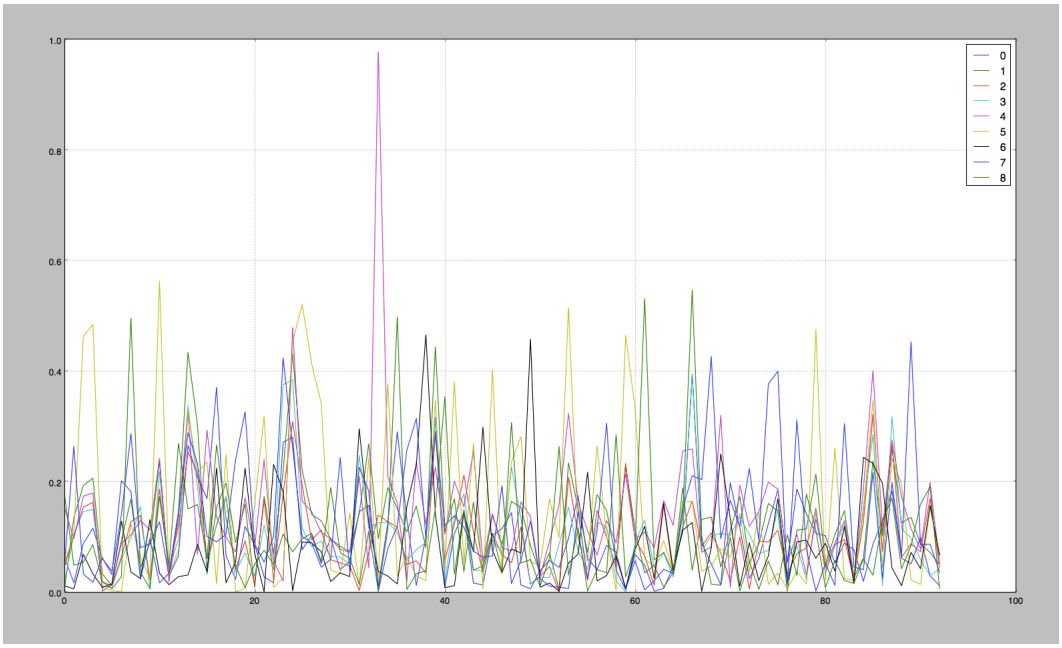

Figure 2: Pseudo-correlation of each feature (x-axis) to each of the 9 classes (different colors)

A few other ideas for feature engineering that I did not get to implement are: unsupervised neural network (different variants of autoencoder), using a different metrics for K-Means than L2, that are more meaningful for count data. After the competition ends, some Kagglers also suggest features such as row sum, row max, number of row that is nonzero, and so on.

3 Models

3.1 Models Used

1. XGBoost: This was likely the most popular model for this competition. I used the version provided by GraphLab, and wrote a scikit-learn wrapper around it for consistency with the rest of the models. Best LB performance: 0.44x

2. Neural Network: I tried a couple implementations, namely Nolearn/Lasagne, with dropout and ReLU and using GPU, and early on Multilayer Perceptron Classifier (which uses Sparse Autoencoder and ReLU). Best LB performance: 0.44x

3. Kernel Support Vector Machine: RBF kernel seems to performs best: Best LB performance: 0.45x

4. Random Forest and Extra Trees Classifier as implemented in scikit-learn: Extra Trees outperforms Random Forest in this task. Using Calibrated Probability (scikit-learn) helps boosting the log-loss significantly. Best LB performance: 0.44x

5. Logistic Regression: I added the quadratic features to help capturing non-linearity

6. K-Nearest Neighbor

After joining team, Nicholas reminded me that simply setting different random seed and rerun models multiple times can help boosting the accuracy. I also did this for a couple models.

3.2 Results

At the end, we have 18 base models, whose performances are depicted in Figure 3. The heatmap is the correlation among the predicted probabilities (for only class 1) of different models. We see some obvious patterns, for example the four "btc" (Boosted Trees Classifier - XGBoost) models I have are highly correlated. On the left, we provide the 5-fold cross validated log-loss, which is very consistent with the public leaderboard. In my experience, most of the time, the public leaderboard score is around 0.013 to 0.02 *lower* than the CV log-loss. This is the case because I retrain the model on all fold of data, which should helps increase the performance of the model.

Table 1 in the Appendix details each of the model parameter, and log loss in 5-fold cross validation, and in Public Leaderboard if possible.

3.3 Tricks Learned

1. The four 'complicated' models perform much better than the 'simple' models. The four complicated ones are: Neural Network, Gradient Boosting Machine (XGBoost), Kernel Support Vector Machine, and Calibrated Random Forest. All of the can get the Log loss down to around .45 or lower. The simpler ones are Linear Models (Logistic, SVM, LDA), AdaBoost, K-Nearest Neighbor.

2. Extra Trees Classifier outperforms Random Forest (in scikit-learn)

3. Calibrated Probability helps Extra Trees, and Random Forest a lot. SVC in scikit-learn already calibrate probability. Neural net with softmax as activation likely does not need calibration either.

4. RBF Kernel works best for SVM

4 Hyper-parameter Tuning

4.1 Grid Search

At the beginning, I simply use Grid Search CV from scikit learn, to search for hyper-parameter. This work fairly wells for models with few parameters to tune: e.g. SVC (only two), Random Forest (only one important parameter). Some of the 18 resulting models are tuned with this method. Grid Search however easily break down when we want to tune more parameters, for example in XGBoost.

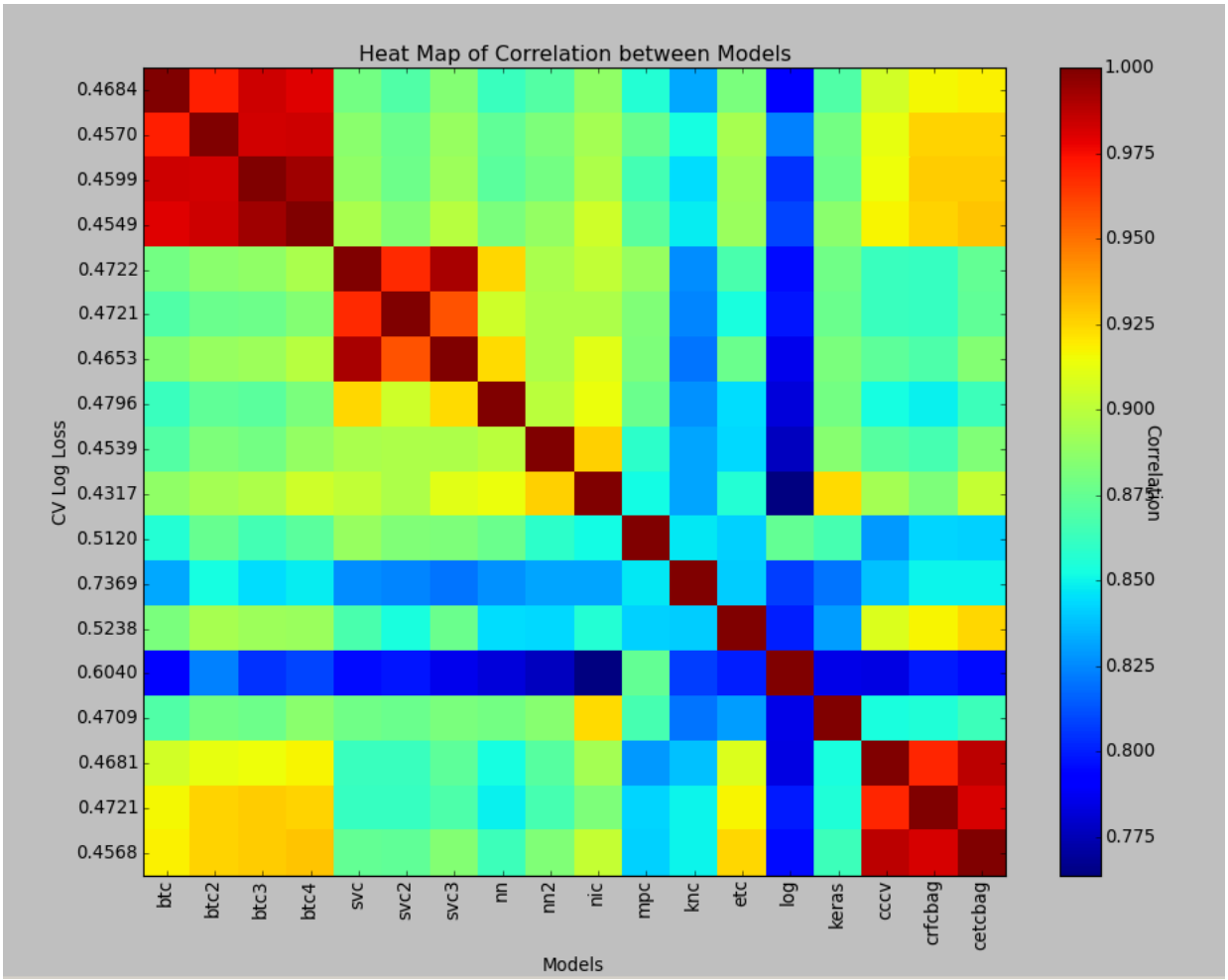

Figure 3: Correlation among all 18 of base models, and their CV log-loss performance. Detail of each model are in the Table in the Appendix.

4.2 Randomized Search

Someone suggested this on Kaggle. Following the paper Bergstra and Bengio 2012, we see that if some of the parameters are not meaningful (but not known to us before hand), then Randomized Search outperforms Grid Search because Randomized Search can evaluate the model at more grid points of the meaningful parameters than Grid Search. I borrow the picture from their paper to illustrate the point:

I used Randomized Search to tune many of the models for this competition.

4.3 Gaussian Process Upper Confidence Bound

The final tool I tried for hyper-parameter tuning is Gaussian Process Upper Confidence Bound, as detailed in Srinivas et al (2009). The idea is at each point, we evaluate the performance of our machine learning model at one set of hyper-parameter, we then pick the next point to evaluate, that maximizes our chance of getting the best set of hyper-parameter. So in Randomized Search for example, we just pick one point uniformly, here we use our prior knowledge of the model, to pick the most likely to succeed set of hyper parameter.

This did turn out to work quite well for some of my models.

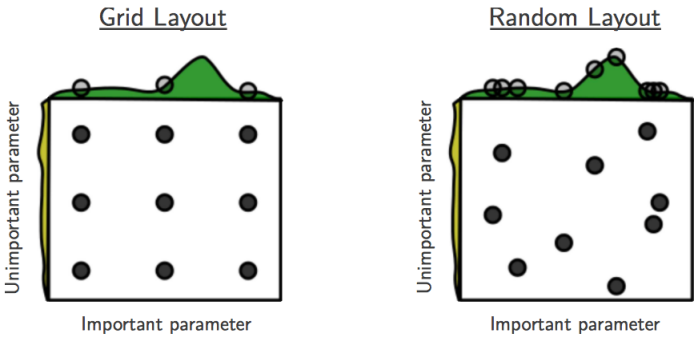

Figure 4: From Bergstra and Bengio 2012. Grid and random search of nine trials for optimizing a function $f(x, y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

4.4 Random Direction

Random Direction is a very simple idea, at each iteration, we have one current best set of hyper-parameter, we then sample the next point from a Gaussian distribution centered around our best set of hyper-parameter so far, and evaluate the model performance at that point. If the performance is better, we pick that point to be our current best, otherwise, we stay with our current best set of hyper-parameter. We keep proceeding that way. Along the iteration, we reduce the variance of the Gaussian distribution to sample from, so that at the beginning we are exploring a wider area, and later on we are exploring in a more local area around our current best set of hyper-parameter.

4.5 Comparison

Following is the comparison of the four different strategies of searching for hyper-parameters. I tried all of them for XGBoost, (Gradient Boosting Machine), and tune parameters as detailed in Table 1. For fast execution, I run the experiment on 1000 data points, repeating the process for all roughly 62000 data points that we have in our dataset. Model performance is evaluated as the log loss on the rest of the dataset.

Hyperparameters	Grid	Random Domain	Description
Max Iterations	$[10, 20, 30]$	Uniform Integer $[5, 35]$	Number of trees to grow
Step Size	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Also called shrinkage
Max Depth	$[5, 7, 9]$	Uniform Integer $[4, 10]$	Max depth of a single tree
Row Subsample	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Portion of rows sampled to build one tree
Column Subsample	$[0.5, 0.7, 0.9]$	Uniform $[0.4, 1]$	Portion of columns sampled to build one tree

Table 1: Gradient Boosting Machine Hyperparameters Grid

I subtracted the average performance of the four strategies from each of the strategies, to account for

the variant in different subset of data. The performance of the four searching strategies are detailed in the box plot in Figure 5. We see that Gaussian Process and Random Direction are performing better. Random Direction performs well though there is more variance to this method (sometimes it gets stuck at a local extremum). Grid Search on the other hand does not perform very well.

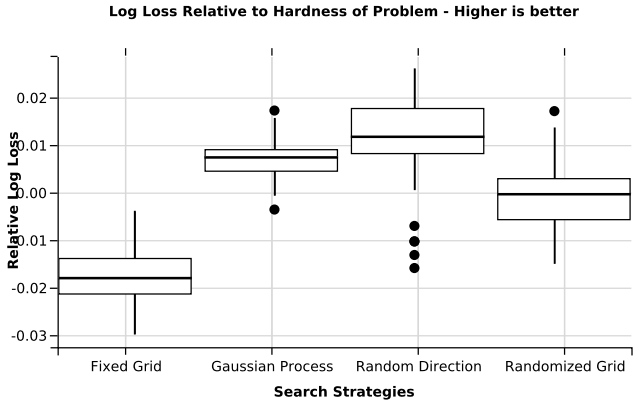

Figure 5: Comparing Four Hyperparameter Searching strategies. Note that this is the LogLoss before taking negative, so higher is better.

5 Ensemble

At the beginning, I tried simple averaging. This works reasonably well. I then learned the trick of transforming my probability estimate using inverse sigmoid, then fitting a logistic regression on top of it. This work better. What I did at the end that work even better, is fitting a one hidden layer neural network on top of the inverse sigmoid transform of the probability estimate. I use the Neural Network implementation from IssamLaradji Github. The hyperparameter for this ensemble was found using Gaussian Process UCB. Following is a rough procedures for my ensemble:

1. For each of the 18 models:
   (a) Obtain the CV-prediction (length 61878 by 9)
   (b) Refit the model on full data, and obtain test prediction (length 144368 by 9)
2. Fit a Neural Network for now the input is a matrix of 61878 by $9 \times 18 = 162$.
3. (Optional) Rerun 1 and 2 for different seeds to further improve the model (around .00x improvement in log-loss).

6 Appendix and Reference

1. James Bergstra, Yoshua Bengio (2009). Random Search for Hyper-Parameter Optimization
2. Niranjan Srinivas, Andreas Krause, Sham Kakade, Matthias Seeger. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design
3. GraphLab Boosted Trees Classifier: https://dato.com/products/create/docs/index.html
4. Paul Duan Amazon Access Winning Solution: https://github.com/pyduan/amazonaccess

5. Issam Laradji Neural Network Toolbox: https://github.com/IssamLaradji/NeuralNetworks
6. Nolearn/Lasagne: https://github.com/dnouri/nolearn
7. Keras: http://keras.io
8. Scikit-Learn: http://scikit-learn.org

Table of Hyper-parameters for base models.

ID	Key	Model Names - Implementations	Parameters	Feature Set	CV LL	LB LL
1	btc	Boosted Trees Classifier - GraphLab	column subsample: 0.9, max depth: 9, max iterations: 154, min child weight: 0.0009765, min loss reduction: 0, row subsample: 0.6, step size: 0.125	original	.4684	
2	btc2	Boosted Trees Classifier - GraphLab	column subsample: 0.6, max depth: 13, max iterations: 378, min child weight: 0.0078125, min loss reduction: 1, row subsample: 0.9, step size: 0.0625	original	.4570	
3	btc3	Boosted Trees Classifier - GraphLab	column subsample: 0.9864, max depth: 9, max iterations: 1503, min child weight: 0.1, min loss reduction: 0, row subsample: 0.5444, step size: 0.0113,	original	.4599	.44611
4	btc4	Boosted Trees Classifier - GraphLab	column subsample: 0.5720, max depth: 8, max iterations: 2468, min child weight: 0.1, min loss reduction: 0, row subsample: 0.4537, step size: 0.011,	original	.4549	
5	svc	Support Vector Classifier - Scikit-Learn	C: 4 γ: 2 Kernel: RBF	TF-IDF	.4722	.46054
6	svc2	Support Vector Classifier - Scikit-Learn	C: 6.29 γ : 16 Kernel: RBF	Log	.4721	
7	svc3	Support Vector Classifier - Scikit-Learn	C: 4.228 γ : 3.334 Kernel: RBF	Text	.4653	
8	nn	Neural Network - Nolearn Lasagne	Layer Sizes: [93, 400, 400, 9] Dropout: [.1, .4, .4] Nesterov Momentum: 0.9 - 0.999 Learning Rate: 0.008 - 0.001 Max Epochs: 400	TF-IDF Standard-ized	.4796	.46421

ID	Key	Model Names - Implementations	Parameters	Feature Set	CV LL	LB LL
9	nn2	Neural Network - Nolearn Lasagne	Layer Sizes: [93, 512, 512, 512, 9] Dropout: [.2, .4, .4, .4] Nesterov Momentum: .9 - .999 Learning Rate: .017 - 1e-6 Max Epochs: 1600	Original	.4539	.44129
10	nic	Neural Network - Nolearn Lasagne	Average of around 30 Neural Network from Nicholas	N/A	.4317	
11	mpc	Multilayer Perceptron Classifier - IssamLaradji Github	alpha: 6.4e-5, hidden layer sizes: 320, activation: relu, algorithm: l-bfgs, max iter: 200, learning rate: constant, learning rate init: 0.5	TF-IDF	.5120	.50381
12	knc	K-Nearest Neighbor - Scikit-Learn	leaf-size: 1000 metric: braycurtis n_neighbors: 32 p: 1 weights: uniform	Original	.7369	
13	etc	Extra Trees Classifier - Scikit-Learn	criterion: gini max_features: 80 n_estimators: 1000	TF-IDF	.5238	
14	log	Logistic Regression	C obtained by LogisticRegressionCV	TF-IDF and its quadratic	.6040	
15	keras	Neural Network - Keras	From Nicholas	N/A	.4709	
16	cccv	Calibrated Extra Trees - Scikit-Learn	method: isotonic cv: 10, n estimators: 300	Original	.4681	.45787
17	crfcbag	Calibrated Random Forest - Scikit-Learn	Averaging 100 times model similar to cccv but for Random Forest, for different random seeds	Original	.4721	
18	cetcbag	Calibrated Extra Trees - Scikit-Learn	Averaging 100 times model cccv for different random seeds	Original	.4568	