Group Project

Data Science & Web Mining (INF131)

Julia Kircher, Nikolas Tzioras, Stefan Coors (Group 1100)

January 29, 2016

Contents

List of Figures

List of Tables

1 Introduction

During the past decades, we witnessed a tremendous increase in computational power and storage capacities allowing new research areas to arise and grow. Those areas are collectively referred to as *computer science*, which is the scientific and practical approach to computation and its applications. Two closely related sub-fields of computer science are *data mining* and *machine learning*. The first one mainly tries to detect patterns within datasets while machine learning deals with *learning* from and making predictions on data. It is divided into *supervised* and *unsupervised learning*. The latter one includes *clustering* and its goals partly coincides with those of data mining, whereas supervised learning has known labels within the data leading to *regression* and *classification* tasks. Our project deals with a *Multi-class Classification* problem. This means that we have a *training dataset* containing observations which are assigned to multiple known classes. The goal of the classification now is to learn, how the observations are assigned to the classes in order to predict the class labels of completely new observations in a new *test dataset*. Obviously, the data analyst tries to maximize the accuracy of those predictions. This can be measured in several ways, e.g. by the *mean misclassification error*, the partition of correct classified observations or, like in this project, by the *multi-class logarithmic loss*.
Section 2 will briefly introduce the given dataset provided by ©Walmart Stores, Inc. on the online platform Kaggle. Furthermore, we will describe how we modified the given dataset for our analysis which is done in Section 3. We start in this section with introducing the different classifier we used during the analysis and continue after some model tuning with the comparison of the results in Section **??** before we conclude our projects by reflecting our findings.

2 Data editing

Starting a machine learning task usually requires data editing from the given raw data. Common reasons for that might be that some features are not helpful for the classification or the general structure of the dataset does not allow a direct analysis. Hence, the analyst needs to perform data editing.

2.1 Dataset

The given dataset by ©Walmart Stores, Inc. includes $95,674$ customer visits in a Walmart Store. Each of the $647,054$ rows corresponds to an item being purchased by a customer during his store visit. Moreover, it includes the following features:

`TripType`	A categorical id representing the type of shopping trip the customer made which we want to predict.
`VisitNumber`	An id corresponding to a single trip by a single customer.
`Weekday`	The weekday of the trip.
`Upc`	The UPC number of the product purchased.

`ScanCount`	The number of the given item that was purchased. A negative value indicates a product return.
`DepartmentDescription`	A high-level description of the item's department.
`FinelineNumber`	A more refined category for each product.

2.2 Merging Visits and Creating new Features

Our task was to learn how Walmart assigned the given *Trip types* to each store visit. This means that we first had to merge all rows containing the same `VisitNumber`. This means, that we shrank our dataset from $647,054$ to $94,247$ rows. Obviously, the feature `VisitNumber` was obsolete since it was just a row index from that on. Hence, we removed it from the dataset. However, since the `Weekdays` are the same for each visit, we only had to replace the string values *"Monday", "Tuesday",...,"Sunday"* with integers $0,\ldots,6$ in order to apply classification algorithms.
However, reducing the dataset would imply an information loss for other features when it is not taken care about the fact that those features have different values for the same store visit. These features are `Upc`, `ScanCount`, `DepartmentDescription` and `FineLineNumber`. Since there are around $90,000$ different `Upc` numbers, we decided to omit this feature since it's classification information is too low. The feature `ScanCount` was modified, so that it contains the sum of all original `ScanCounts` for each visit. Moreover we created for each value of `DepartmentDescription` and `FineLineNumber` new features, containing the sum of all `ScanCounts` for each department and `FineLineNumber` per visit, respectively.
Finally we needed to choose how to deal with missing values. Since we had such a large dataset, we omitted the inclomplete data entries for simplicity reasons.
Hence, our final dataset contained $95,516$ rows (one for each visit) and $5,268$ columns (features) including the true `TripType`.

3 Analysis

Classifying tasks try to predict to which class a new observation belongs, on the basis of a training dataset where the true class assignment is known. In order to verify how good our models classify our observations, we use *10-fold cross validation (CV)*. This concept first divides the dataset at hand into 10 parts. Then, ten models are performed, each one leaving out another of the ten parts of the dataset. Finally the learned classification rules of each model are evaluated on the partition which was left out for the model building process. Moreover, we used the *multi-class LogLoss* as a performance measure for each model during the analysis. This measure is defined as

$$\text{LogLoss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

where N is the number of visits (rows) in the test set and M the number of trip types. Hereby, log() is the natural logarithm, y_{ij} equals 1 if observation i is of class j and 0 if

otherwise. p_{ij} is the predicted probability that observation i belongs to class j. Hence, an ideal (perfect) model which classifies all new observations correctly, would return a Logloss equal to 0 which is hardly achieved in practice. However, the lower the LogLoss, the better the classification model.
We started our analysis with drawing $5,000$ random samples of our dataset. On this smaller dataset, in the following referred to as `train_small`, we first performed our classification model in order to get first results, while keeping the computational algorithm run-time in tolerable limits. During the next sections we will introduce our applied models.

3.1 Logistic Regression

We started our analysis with the given logistic regression model without any further tuning. For multi-class classification problems this model is the multinomial extension of the two-class model which is used for two-class classifications. Its model formula is given by

$$f(c,i) = \beta_{0,c} + \beta_{1,c}x_{1,i} + \beta_{2,c}x_{2,i} + \ldots + \beta_{p,c}x_{p,i}$$
$$f(c,i) = \beta_c \cdot \mathbf{x}_i,$$

where $\beta_c = (\beta_{0,c}, \ldots, \beta_{p,c})$ is the set of the $p+1$ regression coefficients associated with class c and $\mathbf{x}_i$ is the set of the p explanatory variables (features) associated with observation i. As a first step of our analysis, we applied this model to the small dataset without any model adjustment. i.e. we used the default settings provided by the function `LogisticRegression` of the `scikit-learn` package for *Python™*. We achieved a logarithmic loss of 1.89328 in a run-time of 0.5 seconds on our test system which will be described in Section 3.7.

3.2 Naive Bayes

Searching for a simple solution for a complex problem leads us to the *Naive Bayes* classification which is based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Unfortunately, even on the small dataset, our fear that a good classifier on such a complex dataset cannot be as simple as the naive Bayes classifier has perfectly proven true by achieving a LogLoss of 27.13562.

3.3 Decision Tree

Our next model which we applied to `train_small` was a decision tree. This concept uses a tree structure to create a model that predicts the class value based on the feature set, where each interior node corresponds to one feature and each leaf represents a class given the feature values represented by the path from the root to the leaf. Hence, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. A leaf of a full-blown decision tree can contain only a single observation.

Without any tuning, the classifier `DecisionTree` of the `scikit-learn` package delivered a LogLoss of 2.92703.

3.4 Support Vector Machines

The third method we used for our analysis was a *Support Vector Machine (SVM)*. Given two classes which are linearly separable, we can select two parallel hyper-planes that separate the two classes, so that the distance between them is maximal. The region bounded by those two hyper-planes is called "margin", while the *maximum-margin hyper-plane* is the one which lies halfway between them. Those observations which lie on the margin completely determine the two hyper-planes. They are called *Support Vectors*. If the two classes are not linearly separable, one tries to minimize the resulting *loss*. For non-linear classification problems, SVMs are able to perform the *kernel trick* which allows to transform the data in a higher dimensional feature set, where the classes are linearly separable which leads to a non-linear separation in the input space. However, this method increases the generalization error.
As a starting point for our SVM application, we again chose the default settings of the `SVC` function of the `scikit-learn` package. We obtained a LogLoss of 1.99973 for the small training set.

Dataset: `train_small`	LogLoss
LogisticRegression 1	
model param.:	
`default`	1.89328
computation time:	1 sec
LogisticRegression 2	
model param.:	
`C`= 0.4, `solver`='lbfgs',	
`max_iter`= 1,000,	
`multi_class`='multinomial'	1.69224
computation time:	1 min 12 sec
NaiveBayes	
model param.:	
`default`	27.13562
computation time:	12 sec
DecisionTree	
model param.:	
`default`	2.92703
computation time:	13 sec
SupportVectorMachine	
model param.:	
`default`	1.99973
computation time:	1 min 1 sec
GradientBoosting	
model param.:	
`default`	2.60705
computation time:	29 min 4 sec

Table 1: Achieved LogLoss values for the different models on the small dataset. Runtimes on a mid-range Laptop.

3.5 Ensemble Methods

After using some basic models, we decided to use some ensemble methods. Ensemble-Methods construct several predictors ("ensemble of predictors") for a prediction function and use a (convex) linear combination of them to obtain the final aggregated predictor. In detail, those methods use a so-called *base learner* on m weighted datasets which lead

to m different prediction functions. Aggregation of those provide the final prediction function.

3.5.1 Random Forests

As a first ensemble method, we used *Random Forests*, which are a special kind of *Bagging*. Bagging stands for **B**ootstrap **Agg**regation. It combines the results of simple base learner on different bootstrap samples. It is an easy to implement ensemble method which improves the unstable predictions of classification or regression models. Random forests use this concept, where trees serve as base learners. However, this method loses the good interpretability of its base learners. Using the default settings of the `RandomForestClassifier` function which only includes 10 trees, we got a LogLoss of 5.51447 for the small training set.

3.5.2 Gradient Boosting

Our second ensemble method and last model at all for our analysis was *Gradient Boosting*. This method combines gradient descend with additive modeling using weak base learners for minimizing the empirical risk. As base learner, tree stumps are widely used in practice. Tuning parameters are the depth of the trees and also the *shrinkage parameter* or *learn rate* which controls the steps of the gradient descent.
Applying gradient boosting to our small dataset lead to a LogLoss of 2.60705.

Dataset: `train_small`	LogLoss
RandomForest 1 model param.: `n_estimators`= 10, `min_samples_split`=2	5.51447
RandomForest 2 model param.: `n_estimators`= 50, `min_samples_split`=25	2.39214
RandomForest 3 model param.: `n_estimators`= 100, `min_samples_split`=25	2.07358
RandomForest 4 model param.: `n_estimators`= 500, `min_samples_split`=25	1.72604
RandomForest 5 model param.: `n_estimators`= 1,000, `min_samples_split`=25	1.59567
RandomForest 6 model param.: `n_estimators`= 5,000, `min_samples_split`=25	1.59419

Table 2: Achieved LogLoss values for random forest models using a different number of trees.

3.6 Model Selection - Tuning

After looking on the several obtained LogLoss values for each model we decided to concentrate on two models to do model tuning. The first model we chose was that one which performed best with the use of default settings on the dataset `train_small` with 5,000 observations. This was the logistic regression model. Our next step was to tune the model parameters. Here is an overview over the parameters of the logistic

regression model we modified:

`C`	Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. (default=1.0)
`solver`	Algorithm to use in the optimization problem. For multi-class problems, only 'newton-cg' and 'lbfgs' handle multinomial loss; 'sag' and 'liblinear' are limited to one-versus-rest schemes. (default='liblinear')
`max_iter`	Maximum number of iterations taken for the solvers to converge. (default=100)
`multi_class`	Multi-class option can be either 'ovr' or 'multinomial'. If the option chosen is 'ovr', then a binary problem is fit for each label. Else the loss minimized is the multinomial loss fit across the entire probability distribution. Works only for the 'lbfgs' solver. (default= NONE)

We set `C`= 0.4, `solver`='lbfgs', `max_iter`= 1,000 and `multi_class`='multinomial' to improve the LogLoss of 1.89328 to 1.69224 which can be seen in Table 1.
The second model we chose was the random forest, since we believed that tuning this model could be very easy. Therefore, we tuned the following parameters of the random forest:

`n_estimators`	The number of trees in the forest. (default=10)
`min_samples_split`	The minimum number of samples required to split an internal node. (default=2)

For the random forest we found a satisfying `min_samples_split` value of 25 for

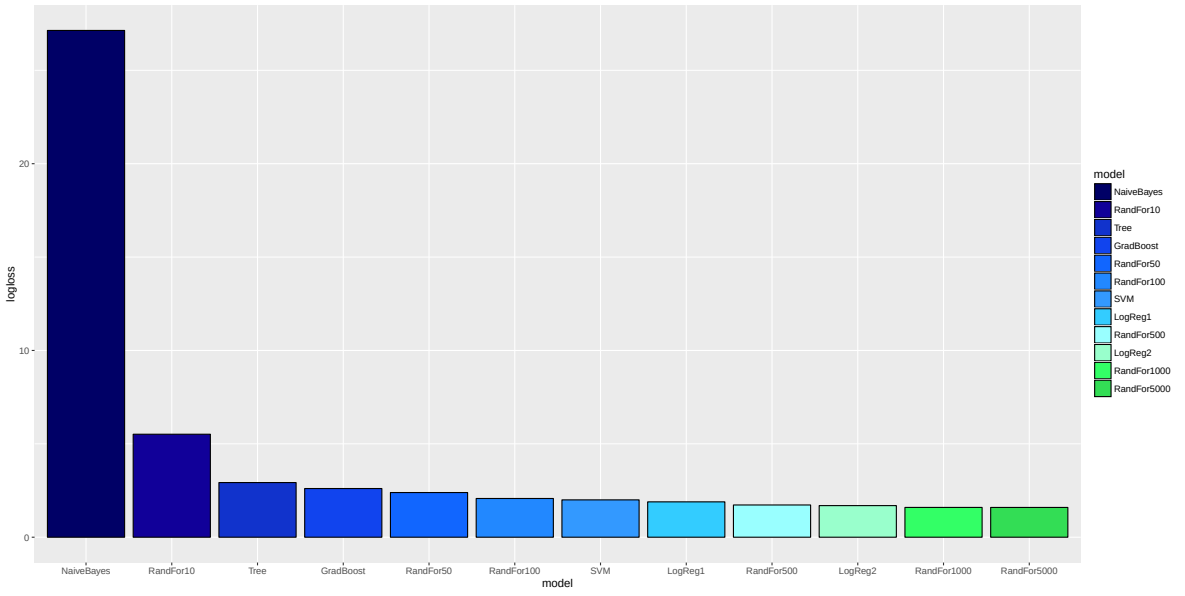

Figure 1: Barplot of LogLoss values on the small dataset

the small train set and increased the number of trees to $(50/100/500/1,000/5,000)$ where $1,000$ seemed for us the optimal trade-off between classification accuracy and model complexity. For $1,000$ trees, we were able to improve the LogLoss from 5.51447 to 1.59567 which can be seen in Table 2. Finally, it was time to apply our two best perfoming models to the complete `train` dataset. Figure 1 shows a barplot of all LogLoss values for the models we tested. The height of the bars are decreasing to the right side which indicates a lower LogLoss, i.e. a better model. Hence, the two random forest models perfromed best overall followed by the tuned logistic regression model. We quickly recognized, that such a high-dimensional dataset containing $95,000$ observations with almost $5,300$ features requires a tremendous computational power. Since our own laptops were not capable of delivering such power, we had the choice whether to reduce our feature set or to find additional computational power. We chose the latter option and rented therefore a server on DigitalOcean.

3.7 Computer System

This server had 20 CPUs and 64GB of memory and was the basis system for our calculations. It was capable of calculating our random forest model with $1,000$ trees on the full train set in 67 minutes. We also tried several smaller servers, but after several obtained errors due to memory overflows, we decided to go with 64GB of ram. Parallelizing the calculation tasks to all 20 cores was simply achieved by setting `n_jobs` to 20.

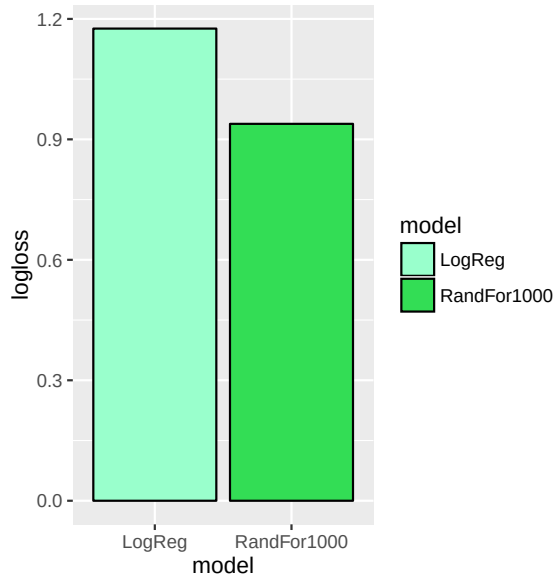

Figure 2: Final LogLoss values on the complete train set.

Dataset `train`	LogLoss
logisticRegression	1.17591
model param.:	
`default`	
computation time:	
RandomForest	0.93865
model param.:	
`n_estimators`$= 1,000$	
computation time:	67 min

Table 3: Achieved LogLoss values, parameters and runtimes of our applied models.

3.8 Achieved minimum LogLoss

Table 3 shows the final achieved LogLoss values of the models applied on the `train` set. Since the random forest model with $1,000$ trees performed similar to the one with $5,000$, we only used the smaller model due to runtime reasons. Moreover, we experienced runtime difficulties with the tuned logistic regression model. As Table 1 indicates, the untuned logistic regression model had a runtime of half a second on the smaller training set on our server, whereas the tuned model had a runtime of 1 min and 12 seconds. This runtime increase did not allow us to run this model on the complete training set. Hence, our final models where the untuned logistic regression model and the random forest with $1,000$ trees.
While the logistic regression model performs surprisingly well with a LogLoss of 1.17591, it is still outperformed by the random forest ensemble model with $1,000$ trees. This model provides the best LogLoss we obtained with 0.93865. Table 3 summarizes the mentioned results and shows the runtimes where available. Figure 2 shows a barplot with the final results.

4 Conclusion and Outlook for further Research

This project was our first approach of dealing with a non-trivial classification problem. During our analysis we tried several models in order to solve the classification task as good as possible. However, theoretical high performance models like gradient boosting or SVMs did not perform as well in their default parameter settings as we previously expected. This might be caused by the fact that a lot more tuning needs to be done. For simplicity reasons we discarded the idea of concentrating on tuning more than doing it manually for some parameters. It would be interesting to see how automated tuning algorithms like *irace* would perform on these models and how far those settings could improve the results.
Moreover, we did only increase the feature set while not applying feature selection methods or dimensionality reduction methods at all except manually removing the variables `VisitNumber` and `Upc`. Applying e.g. a principal component analysis in order to reduce the data's dimensionality while retaining as much as possible of the data's variance could eventually lead to a much faster run-times while achieving a similar LogLoss.
However, holding on to such a high-dimensional feature set showed us the tremendous amount of computational power needed for the model calculations making this whole process even more fascinating.