SD210 Data Challenge
Methods and Results

Amat François
Chalier Yohan
El Harouchi Ali
Slaoui Reda

Final score: 0.69708004995

Prelude

The following report is a chronological report.

Our working flow was pretty linear. First we took a look at the data, trying to understand the issue. Then we tried different classifiers, which performed rather low. We then tried to work on the data structure, cleaning and selecting features. We finally tried with a more advanced classifier to achieve our final score.

Content

Repository

Our code is available on GitHub at https://github.com/ychalier/AlphaShoe.

The two main ipython notebooks we used are also appended to this report. We exported them directly from the .ipynb using Jupyter.

1. Understanding

First of all, we had to understand the whole dataset. By showing the data linked to the following files: `customers`, `products`, `x_train`, `y_train`, `x_test`; we have begun to get familiar with all the features and the link between these files.

The datasets were quite big as we had more than a million data and by merging the different datasets, we happened to learn that we were dealing with a total of 50 features.

We directly, by looking at the whole set of features that our main task would consist of choosing the “best” features which will allow us to train a classifier efficiently.

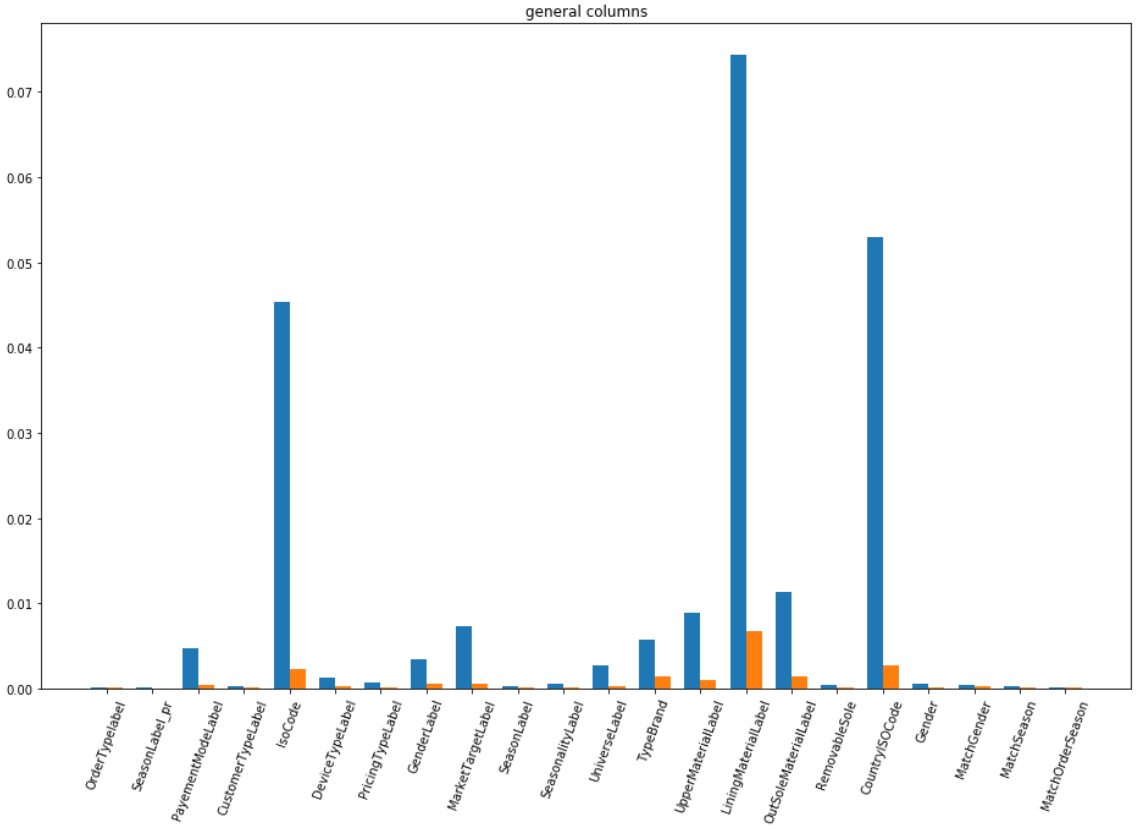
Also, some of the features were strings. We knew that It would have been really difficult to work with such a type. We then had another task, which was: changing these strings into binaries.

2. Statistics

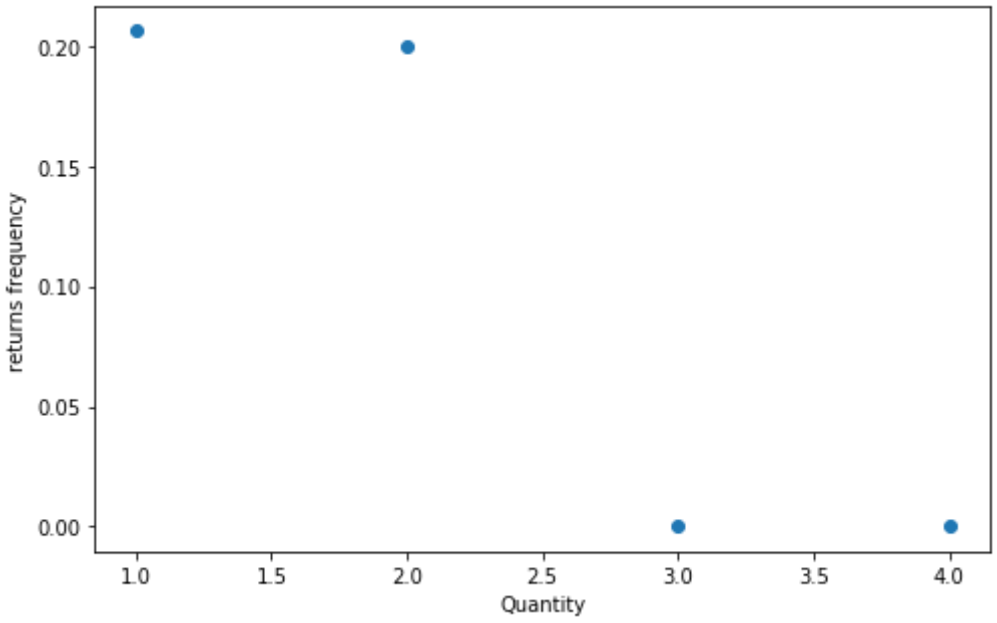
Our second step to understanding the data was to perform some kind of low-level statistics to get an idea of what features would be important to keep. Our objective was double. First we were looking at a possible prior we could unveil. Spoiler alert: results were not specific enough for us to do such thing. The second aim of this statistics study was to test some features to add or to remove.

In order to do this, we computed the return frequency for each value of each column. For example, for the “Gender” feature, the label “Femme” has 21.2% returns, 16.1% for “Homme”, when the “NaN” was around 21.4%. This was performed for every column. Then, in order to get an idea of the importance of the feature, we tried to evaluate its discrimination power, which we evaluated as the relative standard deviation of the feature. We chose this as the standard deviation is a natural way to evaluate the disparity, but sometimes the fact that a column would have so many different values (like “SupplierColor”) made it not persistent. So we divided the standard deviation by the number of values for the feature.

Finally, to visualize the results, we opted for a barchart for the string-like columns. In blue, the standard deviation. In orange, the relative standard deviation.



For the float-like columns, a simple plot was sometimes enough to find a pattern in the results. For example, this graph represents the feature “Quantity”. We can deduce that if more than 3 object are bought at the same time, there is no return.



So was our reflexion. We were slowly starting to understand what we would have to do. As we’ll explain later, we introduced new features based on those statistics. But we soon realized that it would not lead us very far. Our approach was too manual and not well-structured. We ended up computing a more theoretical feature performance from Random Forests.

3. Data cleaning

One of our major work was cleaning our datasets. Actually, we had a lot of features with different types. For example, some of it were integers and some others were strings.

A first cleaning task was to convert `UnitPMPEUR` into a float feature, which was showed in the example notebook. Then we cleaned `SizeAdviceDescription` by replacing the strings with an numerical value corresponding to the actual change in size to operate. For example, if the string advocates to elect a shoe one size lower, we would change it to `-1`. *The idea here is to map the feature to a set of integers while maintaining its semantic properties*.

Then came date-time features. Those were unusable by the classifiers if let as they were. Therefore we have formatted them in order to have integers. For example, for `FirstOrderDate` we have changed the data to the number of the week in the year. The idea was that maybe there was a time in the year when “returner” customers would start buying. For `OrderCreationDate`, we have changed the date by the season and added a new feature with the hour of purchase. Once again, the idea behind that was the existence of a “rush hour”, a time when people make impulsive purchases that would then be more likely to be returned. For `BirthDate`, we have modified the birthdate to the age of the customers.

Finally, the remaining strings features were replaced by adding binary columns to our datasets. For example, for “`Gender`”, we added some columns for every string value of this feature, with 1 as a “yes” and 0 as a “no”. We later tried with a `LabelEncoder` to obtain integer values instead of binaries, but this method performed worse. *This might be due to the fact that mapping labels to integers is generally not isomorphic. For example, it introduces an order that may not be relevant for the feature.*

We have also added some new features:

- `MatchGender`: check if the `Gender` is the same as the `GenderLabel`
- `MatchSeason`: check if the `SeasonLabel_pr` is the same as the `SeasonLabel`
- `MatchOrderSeason`: check if the `OrderSeason` is the same as the `SeasonLabel`

As we later found when computing feature importances, those are quite relevant.

4. First attempt with basic classifiers

Once the data was cleaned, we were able to perform some computation to get some scores. We originally tried with a logistic regression and a decision tree. The first one performed quite good, but the second one was showing too much overfitting.

We then tried with a random forest to counter this phenomenon, using cross validation to determine a maximum depth for the trees. The classifier we finally obtained was rather good, and performed almost as good as our logistic regression, but slightly worse. We ended up doing most of our tests with a logistic regression.

We also wanted to mention that the mask we used to convert string features into numerical features had a different impact regarding the classifier that was used. The binary mask performed better with the logistic regression. The label encoder performed better on the random forest.

The results were ok, as we were able to beat the *admin* on the scoreboard. But at this point even tweaking metaparameters was not improving the score. And as the training score was pretty close to the testing set, we got the idea to work on the data again. It was not represented well enough. So a first idea was using a Principal Components Analysis (PCA).

5. PCA

We used this procedure in order to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The eigenvalue for a given factor measures the variance in all the variables which is accounted for by that factor. High variance means low covariance which means high importance. In other words, if a factor has a low eigenvalue, then it is contributing little to the explanation of variances in the variables and may be ignored as redundant with more important factors.

In our case, we used the PCA algorithm to modify the data as described above. Then, we train with this data a logistic regression classifier in order to see if we obtain at the end a better score than the one found without using PCA.

The problem can come from the fact that PCA is focused on finding orthogonal projections of the dataset that contains the highest variance possible in order to find hidden linear correlations between variables of the dataset. However, nothing guarantees the fact that our data is linearly correlated.

We finally obtained a lower score which led us to avoid using the PCA method. We did not wanted to sacrifice score for computing performance. So we focused on another point, which was that our algorithm was performing low because a strong bias on our training data.

6. Dataset permutations

In our precedent tests, we were only using the first 50000 rows of the data frames, without really knowing if they were relevant in order to construct our classifier. To figure this out, we decided to shuffle the training set, without forgetting to apply the same permutation to `y_train`, in order to have the good answer in the column `ReturnQuantityBin` for each row of the training set. To do that, we concatenated `x_train` with `y_train` and then shuffle them, before separating them again. The dataset resulting from that was then splitted into a training and a testing sets. After that, we trained the classifier following the logistic regression. We did this several times to have an idea of the persistence of the results, and then we kept the classifier giving the best test score. We used this classifier to submit some results, which were significantly improved (about 0.2 points of ROC score).

Not only we shuffled the dataset, we also selected different numbers of rows that will train our classifier. We tried slices of 10,000 to 200,000 rows with a step of 10,000. At each step, we computed the test score to retrieve the slice length that would give the best performance. Our idea was then to trade-off between underfitting and overfitting. Unfortunately, results were not really persistent, and increases in score would be about 0.0001 points.

Now that results were slightly better, we tried again with the classifiers we tested before. And we also tried a new classifier, the support vector machine.

7. SVM

We also tried to use the support vector machine algorithm and see whether we are going to obtain a higher score than the one obtained before. This algorithm consists on finding the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane.

However, as we saw during Machine Learning courses, this algorithm suggests that data is linearly separable which is a priori not the case here. We can use it in non linearly separable data but we have to use a particular type of kernels. In our case, we used it as if our data was linearly separable and as expected, the classifier score was lower than the one obtained before.

The results were then inconclusive. Moreover, it took too much time of execution. So we went back at working on the data, and tried to estimate the features performance.

8. Features importance

When computing basic statistics on the data, we have tried to find the best features in a manual way. Actually, we chose what we thought were the the most relevant features. This approach had its limits as we have no formal proof supporting our choices.

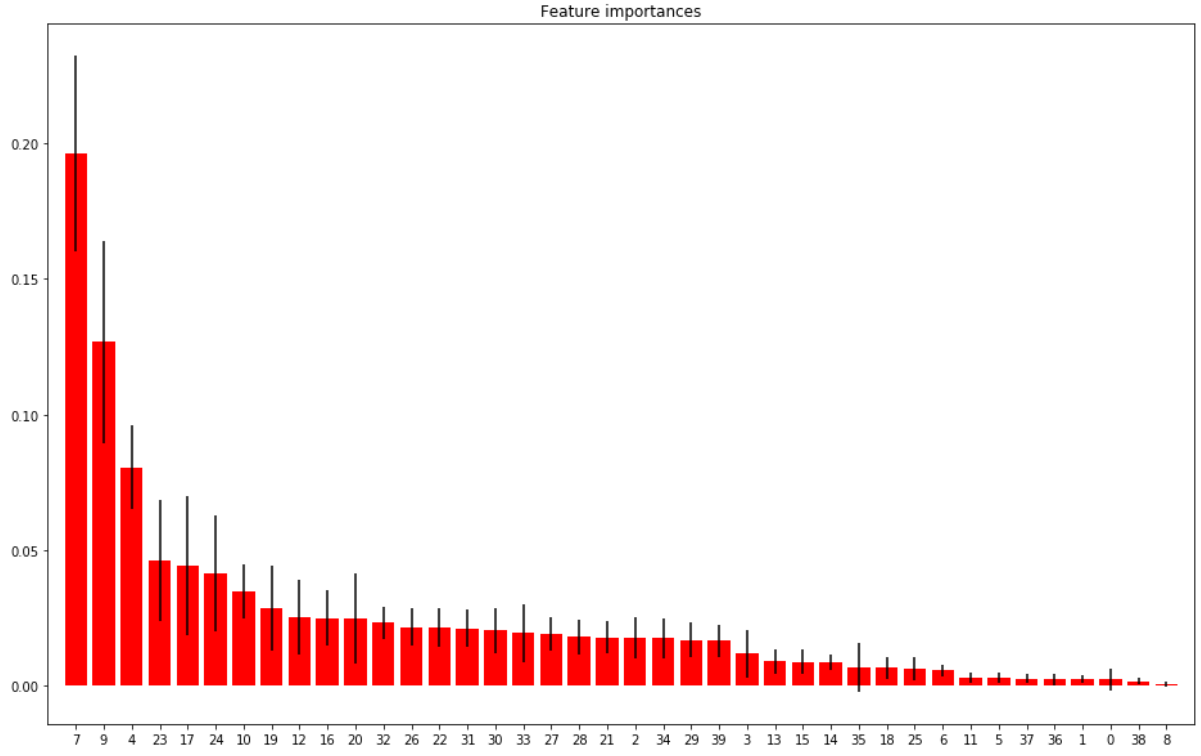
So we decided to compute the features importance in a more formal way. The random forest classifier (which computes a large number of trees), actually implements a lot of what we had in mind. When splitting at a node, the best direction and the best threshold for this split is computed. An inner scheme of features importance is thus built by the classifier. With scikit-learn, this importance (in percent) is available.

Afterwards, we just had to select our features by setting up a threshold. For example, we can take every features which importance pourcentage is above 1%.
By doing this, we succeeded to reduce our features number and keep a similar score to our previous best score. Hence, in terms of efficiency, this method is really interesting as it permits to reduce our time computation. However, it did not reduce overfitting issues.

Here is the 15 top features according to the random forest classifier:

1. `TotalLineItems (0.196264)`
2. `UnitPMPEUR (0.126715)`
3. `IsoCode (0.080465)`
4. `HeelHeight (0.046073)`
5. `UniverseLabel (0.044242)`
6. `PurchasePriceHT (0.041414)`
7. `OrderNumCustomer (0.034733)`
8. `MinSize (0.028464)`
9. `GenderLabel (0.025158)`
10. `BrandId (0.024880)`
11. `MaxSize (0.024448)`
12. `BirthDate (0.022992)`
13. `UpperMaterialLabel (0.021347)`
14. `UpperHeight (0.021340)`
15. `CountryISOCode (0.020999)`

Here are those importances plotted with the error margins:



One may interpret this by stating the existence of a correlation between the top features such as the price, the number of items in the buying cart or the buying country. Some features such as heel heights is surprising in this top, and may be symptomatic of overfitting.

The lack of improvement lead us to some slight cleaning attempts, which helped a bit. But moreover we had to implement a new classifier to get better results.

9. Perceptron

Finally, we used a neural network with a MLP classifier. We tested this classifier against the others basics classifiers in step 4 with two layers of (100,10) neurons. We denoted an improvement of a few percents to the logistic regressor.

In order to get the maximum score, the following parameters of the MLP classifier have been variated: `solver`, `hidden_layer_sizes` and `sample training size`.

First, we must say that we have been very surprised that the computation of the MLP classifier has been extremely slow. In order to test quickly we used external servers. Second, our results show that the best solver is “sgd”, which refers to stochastic gradient descent.

Furthermore, increasing the sample training size enhance the final score, but the computation time explodes (from a few minutes to more than 60 hours). Increasing the number of neurons on the first layer was not improving the result. Increasing the number of layers seems to be very proficient but at the largest scale token the results were lower.

solver	echantillons	hidden_layer_sizes	ROC score
adam	50000	(100, 10)	0.686497819407
sgd	50000	(100, 10)	0.686405943496
lbfgs	50000	(100, 10)	0.690629409463
adam	100000	(100, 10)	0.653439971259
sgd	100000	(100, 10)	0.691128737656
lbfgs	100000	(100, 10)	Nan
adam	50000	(1000, 10)	0.608932650948
sgd	50000	(1000, 10)	0.681890948165
lbfgs	50000	(1000, 10)	Nan
sgd	150000	(100, 10)	0.691470133464
sgd	200000	(200, 10)	0.691520352565
sgd	400000	(100, 10)	0.695817986343
sgd	800000	(100, 10)	0.696591032507

sgd	1000000	(100, 10)	0.697080049900
sgd	50000	(100,100,100,100)	0.684568884792
sgd	80000	(100,100,100,100)	0.688641471032
sgd	100000	(100,100,100,100)	0.689015449398
sgd	1000000	(100,100,100,100)	0.696391366409

Conclusion

Our final algorithm algorithm is really hard to make explicit. Yet, we got an idea of the features of importance during the study.

We have been surprised by the computation time which was pretty important. Actually, we have never worked with a dataset as big as the given one. A lot of our time has been consumed by this computation.

One way of improvement could be a more advanced work on the data. For each column, try to provide an isomorphism to a numerical set, so the classifiers could handle it better.