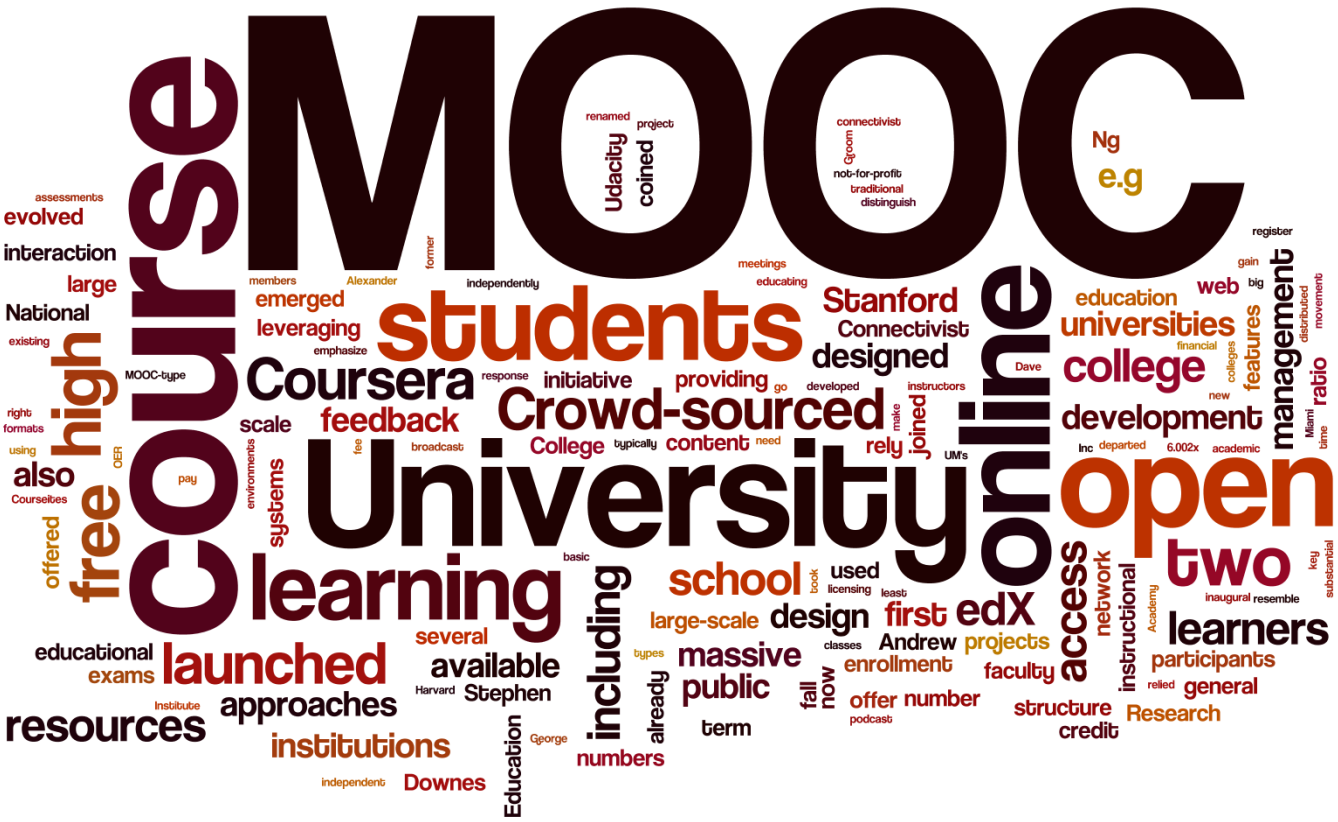
Project Report

KAGGLE MOOC COMPETITION



Participant Name: Soheil Sadeghi (CEE department)

CSE447 – Data Mining

December 2017

Contents

Introduction

In this short report, it is tried to go over the most important aspects of the Data Mining class project that was held through a Kaggle competition. The project was to build a robust classifier that gives high log_loss score on the test data.

Data Preparation

At the beginning, it was needed to import the raw data to the python environment. My first approach was to generate files using read_csv command and then, transform them to numpy arrays. Because I thought that for feature extraction, I need to code some 'For' loops that work much faster on numpy arrays than a DataFrame. However, by some investigation, it turned out that the Pandas module has provided some very powerful tools and commands that make feature extraction much easier than loops over numpy arrays. In this section, I am going over each one of my features and explain how I extracted them from the raw data. Note that features mostly were extracted from activity log file and some were produced from enrollment list file.

Feature extraction

In the enrollment list, we can see that there are about 120K enrollments, which are combinations of user_id's and course_id's. That means the number of courses and students are much less that the count of 120K. By a simple investigation, I figured out that the number of courses are very limited (39 courses) that make the course id a candidate as a feature. Rationally speaking, it makes a lot of sense that the course_id is strongly related to the chance of enrollment dropouts (some courses are more boring than others!). However, the tricky part of transforming course_id's to a numerical feature was to change complicated course_id's (combinations of letters, numbers and signs) to some ordinal labels. Luckily, I could use 'pd.categorical' module that easily did the task. The code is as follows:

```
data_2['course_id'] = pd.Categorical(data_2['course_id']).codes
```

The second feature that is highly weighted is the number of activities per enrollments. In the activity log file, there are bunches of lines that give data for each of enrollments. So, it is needed to first, group data based on their 'enrollment_id' and then, count the number of activities each has. In Pandas library, there is a very powerful tool, called 'groupby' function that group the data based on a certain column index and then, it is easy to implement functions on the well-separated data. Accordingly, I coded the following line to extract activity counts per enrollments as another feature.

```
count = pd.DataFrame({'count':data_1.groupby("enrollment_id").size()}).reset_index()
```

As the next feature, the count of different activities per enrollment was of interest. In order to do so, the activity log got split to different sub-dataframes for each type of activities, and then, each of these sub-dataframes got grouped with respect to the 'enrollment_id' and the number of rows were counted. This process was held seven times for navigate, access, problem, discussion, wiki, page_close, and video activity types. At the end of classification task, it turned out that the majority of these features were highly effective on the splitting power of the classification. However, some of those such as wiki were not that beneficial.

As one of the most weighted features, the entire time spent by each enrollment was considered. This one was one of the most complicated features for me to extract. As the first step toward this goal, I used a very helpful module in Pandas, called datetime module that can easily understand the 'time' column of the activity log. I stored each parts of the 'time' dataframe of the activity logs in separate columns, such as 'month', 'day', 'hour' and 'year'. These columns were used several times to draw out useful features later. To find the activity duration for each enrollment, some complications should be taken care of. First, there was many enrollments that have activities in different days that disables us to easily find the duration by subtracting the last time by the first one for each enrollment. Because the activity times were cut by the last activity at each day and started again on another day. To address this problem, I followed two approaches. The first approach was to group the dateframe by 'enrollement_id' AND 'day' indexes at the same time and subtract the maximum and minimum 'hour's. This new data column is called 'time_diff' in the

model. Next, since there was many activities that happened in less than one hour, I replaced days with zero difference between max and min of hours with one. Finally, I added up the hour differences for each 'enrollment_id' and labeled them as a new feature column called 'entire_time'. The second approach that is more exact is to use diff() function that can easily find durations between activities. Specifically, the dataframe got grouped by 'enrollment_id' and 'day' column and then diff() function applied on their 'time' column. Then, since this new column is a 'time' data and cannot be directly used as a numerical feature, the column was converted to seconds by .dt.total_seconds() function. Finally, to sum up durations for each enrollment, the sum() function was applied on each 'enrollment_id'. Please note that the process is existed in the code and can be understood better by checking the code itself. By the way, both approaches gave the same level of performance, however, since the second method was more accurate, that one was used for the final model.

```
data_1['time_diff'] = data_1.groupby(['enrollment_id','day'])['time'].diff()
data_1['time_diff'] = data_1.time_diff.dt.total_seconds()
entire_time =
pd.DataFrame({'entire_time':data_1.groupby('enrollment_id')['time_diff'].sum()}).reset
_index()
entire_time = entire_time.fillna(0)
```

The next feature is the number of days an enrollment has spent on an online course. It was an easy feature to capture, using the groupby() function. Some more implicit time history information were also used as features for the classification. The most straight-forward one is the standard deviation of the times spent per day by an enrollment. For instance, imagine one enrollment has spent five days and in each, studied 1, 3, 2, 0.5 and 0.1 hours. The time series has a high variance and that may imply the student is not very determined about the online course (irregular studying duration). So, the standard deviation of these five numbers has been used as a feature. To extract this, the std() function which is embedded in the Pandas library, was applied on the 'time_diff' column that is grouped by the 'enrollment_id'. For those enrollments that just have one element in the time difference column, std() function gives NaN. So, NaN's were replaced by a large number, which suggests the student was not consistent in studying.

The second implicit feature is the trend of the times spent in each day by each enrollment. In the example explained in the last feature, it can be figured out that the user is spending less time each day, that means he or she is getting bored or less interested to the course gradually. That may imply that he or she has a high chance of the course dropout. Therefore, the slope or the trend of this time series is a meaningful feature. However, extracting this feature was a complex task for me. To do so, a new library was imported, called statsmodels.api that does regression and returns the polynomial coefficients, based on the polynomial order given by the user. Next, a function was defined to apply linear regression on a Pandas.series and return the second polynomial coefficient, called gen_lin_reg_coef() in the code. Finally, the function was applied to the time different column that is grouped by the 'enrollment_id' and the function returned a column data of the slopes. However, as the last one, for the enrollments with just one time difference, the function cannot find a polynomial coefficient and return NaNs. So again, the replace() function was used to solve this problem. It is worth to mention that in contrast of what I expected, this feature did not improve the classification accuracy considerably. However, it still was used and added the total score.

Finally, after extracting all necessary features, the feature dataframe got standard scaled to be ready for feeding to the classifying algorithms.

Further in the process, some meta_features also were added to enhance prediction power of the classifiers. Those will be explained later in the report.

Models, results and challenges

At the beginning, I tried different classifiers that were covered in the course and those existed in the Scipy library. As it was predictable, some algorithms worked better by the scaled data. However, for tree based classifier (trees, forests and boosting), scaling was not very effective. Before diving into the mechanisms of the classifiers, note that for the classifiers' performance evaluation, the given data was split in two parts (60% for training and 40% for the test). Each algorithm was trained on the training dataset and was evaluated on the test data. Here the models that were used are explained briefly and the quality of result are mentioned.

Preliminary Classifiers

The first algorithm I used was K nearest neighbors from Scipy. The algorithm is not really a classifier, but an estimator that guess a new data's label based on the majority of its surrounding datapoints. Consequently, the algorithm is sensitive to the number of neighboring datapoint that should be taken into account. By a simple parameter study, it turned out that the best result is found when K = 3. The algorithm was one of the most efficient

algorithms, however, the performance needed to improve considerably. Rather that, the predict_proba() function gave values very close to zero and one when used on a KNN estimator directly. To solve these issues, as a possible solution, this algorithm was mixed with the bagging classifier from the sklearn module. Bagging classifier repetitively train a base estimator on a given data and finally predict based on the mean of all predictors. The combination of KNN and bagging worked slightly better, however the computational time was relatively high. Therefore, other algorithms were tried.

The second classifier model that was tried is the support vector machine (SVM). SVM's objective function is to maximize the bandwidth gap between binary classes. Since the algorithm calculates the cost function based on the distances, the algorithm is highly sensitive to the feature scaling. The algorithm has several sensitive parameters that can be tuned, such as the type of kernel. However, spite of the parameter tuning, the performance of SVM on our data was not so attractive. One possible source of inaccuracy was the bias on prediction probabilities caused by the type of classification that can be handled by calibration. This procedure is explained later. To sum up, I did not use SVM results in my final model, because of its high computational cost and weak results.

As the next possible classifier, Gaussian Naive Bayes (GNB) classifier was implemented that despite its very fast implication, suffered from the weak prediction power. Therefore, the calibrated bagged version of it was tuned to enhance its performance. The algorithm gave moderate results and its prediction was used later for meta-classification trials.

In case of SVM and GNB, it was assumed that one possible source of inaccuracy could be the high dimensionality of the input data. To address this problem, some dimensionality reduction algorithms were applied, such as Principal Component Analysis tool (PCA), selectKBest and their combination using FeatureUnion(). Different numbers of components and selections can be chosen in each of these reduction algorithms. However, comparing the reduced input results against the original input, it was deduced that the dimensionally reduction is not an improving technique for our problem.

The next approach was to use MultiLayer Perceptron classifier (MLP) which is a form of Neural Network based classifiers that is provided by Scipy. In this function, coder can define several layers and the kernel function that convert data from each layer to the next level. These parameters were tuned and the performance was evaluated on the test data. Unfortunately, this form of neural network classifier that is sequential and use one kernel for all layers was not a successful approach and gave imprecise predictions.

As the final sets of classifiers, tree based classifiers were employed, such as Gradient Boosting, Random Forest and ADABoost with Extra Trees classifiers. luckily, these sets of classifiers showed very desirable performances. However, they were needed to be accurately tuned. Boosting based trees were slightly more attractive than the classical random forrest, since they put emphasize on the unpredicted data for generation of the new trees. Consequently, these sets of classifiers were used in the final model.

In the end of individual model selection phase, some less common algorithms were also tried, such as Gaussian Process Classifier and Quadratic Discriminant Analysis, which gave moderate to weak performance and were not employed in the final model.

Ensemble classifiers

As it was discussed in the class, the most pioneer and winning classifiers are ensembles of classifiers. Accordingly, in the second phase of the project, I started to build ensembles of many classifiers that were discussed before. Hereafter, these ensembles are explained and challenges are discussed.

The first ensemble algorithms that I tried was the voting classifier. This algorithm is a very basic ensemble method that uses the predictions of different individual classifiers and mix them (basically average them). The way that I have used was first, bag individual classifiers (using Bagging Classifier) and then, input bagged results to a voting classifier for ensemble. A very useful tuning parameter in the voting function is the weights that allows you to put different votes for different individuals in the pipeline. The designed pipeline was extremely time consuming and did not show a significant boost in the performance. Therefore, I decided to design my own voting classifier that is slightly different from a simple averaging of probabilities. The customized ensemble classifier firstly find the mean of results, and then check if the mean probability of a datapoint is higher than (for instance) 0.85, then boost it to the maximum prediction probability among individual classifiers and also if the probability is less than 0.15, then does the opposite. By this logic, I meant to give more confident to the prediction results for those datapoint that the majority of algorithms agreed on their labels. However, the customized ensemble algorithm gave worse results and fully disappointed me to use voting for ensemble classifiers. The reason for worse result is that by manipulating predictions, the algorithm may be strongly confident for those outliers that all individual classifiers labeled wrong, which worsen the result exponentially (since the log_loss function is used). The part of code that made my customized classifier in shown below:

```
prob['mean'] = prob.mean(axis=1)
prob['prob_guess'] = prob['mean']
max_vals = prob[['prob_KNC3','prob_XGB']].max(axis=1)
min_vals = prob[['prob_KNC3','prob_XGB']].min(axis=1)
prob['prob_guess'] = np.where(prob['mean'] >= 0.65, max_vals, prob['prob_guess'])
prob['prob_guess'] = np.where(prob['mean'] < 0.35, min_vals, prob[‘prob_guess'])
```

Another struggle to make predictions more accurate was to calibrate predictions considering the biased values each gives based on their structure. This function filters the prediction probabilities based on the reliability curves for each types of classifier. For this purpose, the following webpage was very instructive:
http://scikit-learn.org/stable/auto_examples/calibration/plot_calibration_curve.html#sphx-glr-auto-examples-calibration-plot-calibration-curve-py
By the knowledge of all trials explained before, it was concluded that two enhancements should be taken place. The first is to add some Meta Features that can capture some unseen discriminative characteristics that my current features are not able to show. The other field of enhancement was to find some high-level classifiers that are designed for big data and are both fast and accurate. In the next phase, these two struggles will be explained.

Meta Features

By investigating in the web, it was figured out that a good approach to draw out implicit meta-features is to use clustering methods. Clustering is a data mining approach that group unlabeled data by their proximity (sometimes, their similarity). Thus, they can find some hyper dimensional interrelations between datapoint that are not obvious by their appearance. Two clustering methods, K-Mean and DBSCAN were implemented on the database. For the K-Mean clustering, the number of clusters is a sensitive parameter that should be tuned. By a parameter study, the best K is found 10 and used for the meta_feature generation. In case of DBSCAN, the minimum sample points in proximity was tuned and the best value was found to be 50. Other than these two, the physical distances between datapoint were computed through the KNN algorithm and three other meta_features were produced as sum of the distances of 1- two closest, 2- four closest and 3- six closest datapoint for each datapoints. The following code shows how the task is done for meta_features:

```
kmeans = KMeans(n_clusters=10, random_state=0).fit(X_new)
clusters_kmean = kmeans.labels_
DB = DBSCAN(eps=0.3, min_samples=50).fit(X_new)
clusters_DB = DB.labels_
A = list(KNC_log.kneighbors(np.log(10+X_new),9))[0].tolist()
dist = pd.DataFrame(A).reset_index()
dist2 = np.array(dist[[0,1]].sum(axis=1))
dist4 = np.array(dist[[0,1,2,3]].sum(axis=1))
dist6 = np.array(dist[[0,1,2,3,4,5]].sum(axis=1))
```

High-level classifiers

In order to find high-level tools for classification task, some complex and powerful modulus were checked like Keras and XGBoost. Keras is a very flexible and easy to use library for different types of Neural Networks. It gives the flexibility to define different kernels for each layer and also employ convolution layers. In my application, I tried both sequential and convolutional networks to classify data; however, the algorithm was highly prone to overfitting. The log_loss on trained data was in the order of 0.2; however, the same classifier gave log_loss of 0.37 on the test data. Some regularization tools were used to address this problem. Finally, after tuning parameters, the algorithm resulted 0.34, which was an enhancement with respect to all preliminary approaches taken before. However, it was not still close to the pioneer results in the leaderboard and I tried to find even better algorithms. Keras classifier can be found in version 4 of the code, exists in the package.
As mentioned before, tree based algorithms showed relatively better results comparing to other types of classifiers, so a high-level tree based classifier was employed, called XGBoost. XGBoost is an eXtreme case of boosting algorithm, which gave by far the best performance in my final solution. The algorithm is very time cost effective and provides various parameters that let to control on the performance. That means the parameter tuning phase is very involving and took a lot of time for me to be done. The following webpage helped me a lot to tune parameters in a careful manner:
https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
The log of different performances for parameter tuning is provided in the package as performance2.xls file. Finally, the following configuration showed the most desirable performance:

```
XGB = xgb.XGBClassifier( learning_rate =0.1, n_estimators=133, max_depth=6,
min_child_weight=9, gamma=0.0, subsample=1.0, colsample_bytree=0.6, objective=
'binary:logistic', reg_alpha = 10, nthread=4, scale_pos_weight=1, seed=27)
```

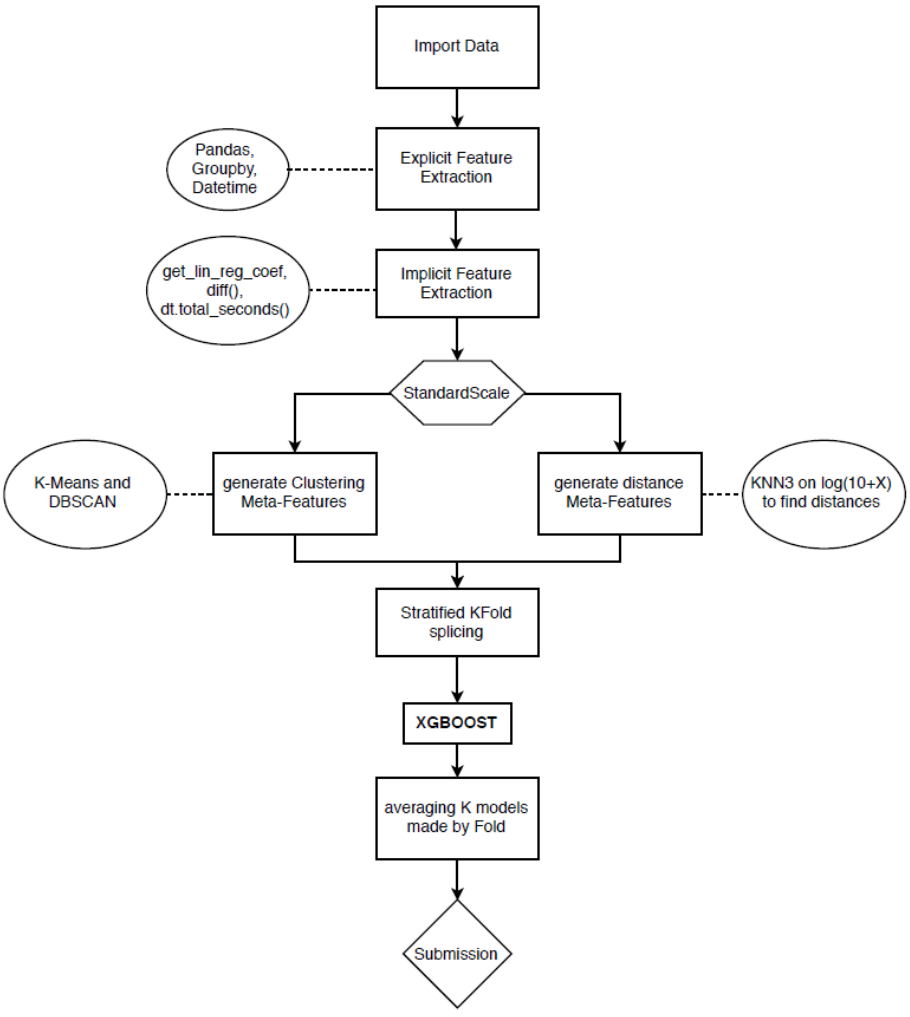
A good feature that is placed in XGBoost package is to get an evaluation set and display accuracy metric values on both training and test datasets for each estimation. This feature helped a lot for parameter tuning.

When the XGBoost classifier was fully tuned, a more efficient cross validation approach (better than simple splitting by 60%, 40%) was needed to produce final models. Different methods, such as folding, Kfold and Stratified Kfold were applied and the mean of K classifiers were used to evaluate their performance. Surprisingly, stratified Kfold enhanced the quality of results considerably and therefore, was used as a final approach to produce a robust and unbiased final model.

By the way, just to mention all my struggles, I wanted to explain that once I was thinking that since the binary labels are highly imbalanced, some more avant-garde approaches can be useful, such as anomaly detection. I could not deeply investigate anomaly detection methods in my project; however, to handle imbalanced data in the XGBoost algorithm, there is a parameter, namely scale_pos_weight, that places different weights for positive and negative classes. A parametric study on this parameter showed that the best value is 1.0 that means the varying weights for classes do not improve the accuracy of results.

Finally, to aggregate classification results of the leading models, linear and geometric averaging of the prediction probabilities were held. Four models were selected as the leading models to be integrated, which were XGBoost, KNN3, KerasNN and ADABoost with Extra Trees. However, after tuning weights, it was deduced that the best result is just given when all weights are on the XGBoost model. Hence, my final submissions that gave the most accurate results are outputs of my XGBoost model.

To depicting my designed pipeline for the final model, the following flowchart could be demonstrative.



Failed trials

At the end, some of my failed struggles are breifly mentioned hereafter.
In order to build a meta classifier, once I fed a bunch of predictions estimated by many individual classifiers as input features to my XGBoost classifier and see whether the results get better or not. However, the effect of feeding predictions was not enhancing and was highly prone to overfitting. It can be interpreted that in this situation, the model put an overt confident on the predictions of other models, which themselves maybe flawed. Consequently, this approach was not used in the final pipeline. The script of this try can be found in the version 3 of the main file.
Another approach was to make subsample sets with equal population of 0 and 1 labels. With this method, it was aimed to train classifers that are equally powerful to estimate both labels (treat imbalanceness). A sub function was written (can be found in version 4 of the main file) to do the job and produce equal sample bags and then, using this dataset, a classifer was trained and this process repeated multiple times. However, this approach was also ineffective and could not enhance prediction scores. The reason could be the small size of dataset for each classifier and oversensitivity of the models to 0 cases that caused weak prediction capability.

Conclusion

In this project, we were exposed to a real application of data mining and tried our best to enhance our model performance. We learned that some policies are more influential and worth to take, such as designing a pipeline, instead of an individual model, or adding meta-features from byproducts of individual models. An serious exposure to python environment also was a valuable catch for me in this project. As a civil engineer who are trying to apply data science technique in his research, this project and in general, the course helped me a lot to get familiar with the field and enjoy from it.