CS 334 Machine Learning Final Project Report

Zirui Deng, Yinfeng Wang

Abstract

In this project, we worked on the identification of credit card frauds. We used a dataset from Kaggle. The dataset is extremely unbalanced, as only 0.172% are fraud transactions. We chose recall score and area under the precision-recall curve as our model assessment metrics. As the dataset from Kaggle had already been preprocessed using PCA, we constructed a heatmap that showed low correlation between features. After that we split 30% of the dataset as our test set and kept the same ratio of fraud transactions in the training and test set.

We applied different machine learning classification models to the dataset and then computed metrics scores and plotted precision-recall curves of the models. We compared the results of the models and analyzed the reason why some models perform better than others for this project from aspects of our dataset and model algorithms. In the end we also reflected upon the limitations of our approach and acknowledged that future work could be done to make our results more precise and comprehensive.

Introduction

In this project, we aimed to address the problem of identifying credit card frauds. Nowadays people use credit cards anywhere they go. While credit cards certainly are a convenient way to make payments, for the past few years there have been numerous cases of fraudulent credit card transactions, and it is very important that credit card companies be able to recognize those frauds so that customers are not charged for items they did not purchase. Motivated by this reason, we decided to work on a project to detect credit card frauds.

In the project, we were dealing with a classification problem using the dataset collected from Kaggle, with class ‘0’ for normal transactions and ‘1’ for credit card frauds. Considering the unbalanced nature of the dataset, our approach to the problem was, after data preprocessing and feature selection, to experiment with different machine learning classifiers and then use these models as base learners to explore ensemble methods and see if they indeed work better according to our model assessment strategy.

The model assessment metrics used to evaluate the models should be suitable for the distribution of our dataset. Conventional measurements such as accuracy of prediction would not be sufficient to distinguish between models, as almost all of them have an accuracy above 99%. Recall that since 99.8% percent of our data were labeled ‘0’, every model would have lots of true negatives (TN). We concluded that we should evaluate models by the true positives (TP), which means the area under the ROC curve would not be sufficient to evaluate models as well. So

instead of metrics learned in class, we decided to use the precision-recall metric. The formula for precision and recall are

$$P(Precision) = \frac{TP}{TP+FP} \text{ and } R(Recall) = \frac{TP}{TP+FN} .$$

We observed that neither precision nor recall is influenced by TN, and both of them evaluate TP. This is exactly why we deemed precision-recall metric suitable for our data. Besides area under precision-recall curve (AUPRC), which is proposed by Kaggle, we also employed precision score, recall score and average precision score (AP) as other measurements of the model performances.

Background

The original dataset has been collected and analyzed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Université Libre de Bruxelles) on big data mining and fraud detection. The data that we used contains only numeric input variables which were the result of a PCA transformation previously done by others. Due to confidentiality issues, the original features and more background information about the data are currently disclosed from the public. We noted that there are many approaches done by others on Kaggle. The one with the most upvotes was done by Janio Martinez. He approached the problem by random undersampling with a 50/50 ratio of fraud and non-fraud and achieved a 93% recall score using Logistic Regression and SVM.

Methods

We decided to try out various classification models that were covered in class and compare their effectiveness in order to choose the optimal models for our task. We first took a look at KNN, which is the most basic classification models. Then we ran a Decision Tree classifier, another relatively basic learner, to see if it would do better than KNN. We also tried to see if a binary linear classifier could do the job, and we used Perceptron accordingly; we also picked Logistic Regression which is considered a generalized linear model for binary classification. Afterwards, we moved on to some more advanced models with higher-level classifiers. We first chose Complement Naive Bayes. Naive Bayes assumes that the features are independent, which is suitable for our case as we have checked previously. The reason we used Complement Naive Bayes in particular was that this adaptation of the standard multinomial Naive Bayes algorithm particularly fits for unbalanced datasets. SVM was also selected because this model uses margin for classification as well as optimization to find solution with fewer errors, and it has been proven to be one of the most efficient models.

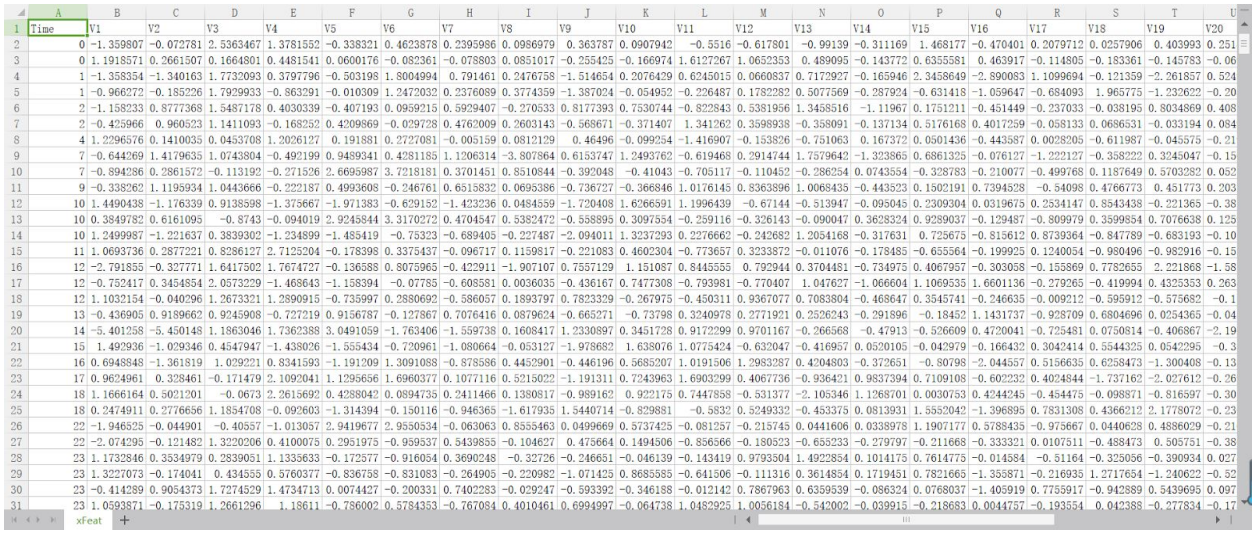
Afterwards, we proceeded to try some ensemble methods which essentially take advantage of several learners to solve the same problem. We first employed two classic ensemble methods, Random Forest and Bagging with Decision Tree as the base learner. Here we decided to run two

models with similar tree structures so that we could make a reasonable comparison between these two ensembles as well. Other ensemble methods we picked were Voting and XGBoost. We used Logistic Regression, Decision Tree and Complement Naive Bayes as base learners for Voting with adjusted weights so as to see if Voting could indeed improve the performances of the three base models to a discernible degree. We also chose XGBoost as this is one of the most popular algorithms that is widely used by those who excelled in Kaggle competitions. We were curious to see if XGBoost could really generate results that would stand out among other models. Finally we turned to Neural Network. In particular, we chose Multilayer Perceptron (MLP). We wanted to see if using MLP could be a noticeable upgrade from Perceptron.

Experiments/Results

The dataset contains transactions made by credit cards in September 2013 by European cardholders. It presents transactions that occurred in two days, during which there were 492 frauds out of 284,807 transactions included in the data. The dataset is highly unbalanced, with the positive class (frauds) accounting for only 0.172% of all transactions.

It contains variables that are the result of a PCA transformation. Features V1, V2, ... V28 are the principal components obtained from PCA; the only variables that have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' denotes the seconds elapsed between each transaction and the first transaction in the dataset. 'Amount' is the transaction Amount. The 'Class' column is the response variable, and it takes value 1 in case of fraud and 0 otherwise.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20
2	0	-1.359807	-0.072781	2.5363467	1.3781552	-0.338321	0.4623878	0.2395986	0.0986979	0.363787	0.0907942	-0.5516	-0.617801	-0.99139	-0.311169	1.468177	-0.470401	0.2079712	0.0257906	0.403993	0.251
3	0	1.1918571	0.2661507	0.1664801	0.4481541	0.0600176	-0.082361	-0.078803	0.0851017	-0.255425	-0.166974	1.6127267	1.0652353	0.489095	-0.143772	0.6355581	0.463917	-0.114805	-0.183361	-0.145783	-0.06
4	1	-1.358354	-1.340163	1.7732093	0.3797796	-0.503198	1.8004994	0.791461	0.2476758	-1.514654	0.2076429	0.6245015	0.0660837	0.7172927	-0.165946	2.3458649	-2.890083	1.1099694	-0.121359	-2.261857	0.524
5	1	-0.966272	-0.185226	1.7929933	-0.863291	-0.010309	1.2472032	0.2376089	0.3774359	-1.387024	-0.054952	-0.226487	0.1782282	0.5077569	-0.287924	-0.631418	-1.059647	-0.684093	1.965775	-1.232622	-0.20
6	2	-1.158233	0.8777368	1.5487178	0.4030339	-0.407193	0.0959215	0.5929407	-0.270533	0.8177393	0.7530744	-0.822843	0.5381956	1.3458516	-1.11967	0.1751211	-0.451449	-0.237033	-0.038195	0.8034869	0.408
7	2	-0.425966	0.960523	1.1411093	-0.168252	0.4209869	-0.029728	0.4762009	0.2603143	-0.568671	-0.371407	1.341262	0.3598938	-0.358091	-0.137134	0.5176168	0.4017259	-0.058133	0.0686531	-0.033194	0.084
8	4	1.2296576	0.1410035	0.0453708	1.2026127	0.191881	0.2727081	-0.005159	0.0812129	0.46496	-0.099254	-1.416907	-0.153826	-0.751063	0.167372	0.0501436	-0.443587	0.0028205	-0.611987	-0.045575	-0.21
9	7	-0.644269	1.4179635	1.0743804	-0.492199	0.9489341	0.4281185	1.1206314	-3.807864	0.6153747	1.2493762	-0.619468	0.2914744	1.7579642	-1.323865	0.6861325	-0.076127	-1.222127	-0.358222	0.3245047	-0.15
10	7	-0.894286	0.2861572	-0.113192	-0.271526	2.6695987	3.7218181	0.3701451	0.8510844	-0.392048	-0.41043	-0.705117	-0.110452	-0.286254	0.0743554	-0.328783	-0.210077	-0.499768	0.1187649	0.5703282	0.052
11	9	-0.338262	1.1195934	1.0443666	-0.222187	0.4993608	-0.246761	0.6515832	0.0695386	-0.736727	-0.366846	1.0176145	0.8363896	1.0068435	-0.443523	0.1502191	0.7394528	-0.54098	0.4766773	0.451773	0.203
12	10	1.4490438	-1.176339	0.9138598	-1.375667	-1.971383	-0.629152	-1.423236	0.0484559	-1.720408	1.6266591	1.1996439	-0.67144	-0.513947	-0.095045	0.2309304	0.0319675	0.2534147	0.8543438	-0.221365	-0.38
13	10	0.3849782	0.6161095	-0.8743	-0.094019	2.9245844	3.3170272	0.4704547	0.5382472	-0.558895	0.3097554	-0.259116	-0.326143	-0.090047	0.3628324	0.9289037	-0.129487	-0.809979	0.3599854	0.7076638	0.125
14	10	1.2499987	-1.221637	0.3839302	-1.234899	-1.485419	-0.75323	-0.689405	-0.227487	-2.094011	1.3237293	0.2276662	-0.242682	1.2054168	-0.317631	0.725675	-0.815612	0.8739364	-0.847789	-0.683193	-0.10
15	11	1.0693736	0.2877221	0.8286127	2.7125204	-0.178398	0.3375437	-0.096717	0.1159817	-0.221083	0.4602304	-0.773657	0.3233872	-0.011076	-0.178485	-0.655564	-0.199925	0.1240054	-0.980496	-0.982916	-0.15
16	12	-2.791855	-0.327771	1.6417502	1.7674727	-0.136588	0.8075965	-0.422911	-1.907107	0.7557129	1.151087	0.8445555	0.792944	0.3704481	-0.734975	0.4067957	-0.303058	-0.155869	0.7782655	2.221868	-1.58
17	12	-0.752417	0.3454854	2.0573229	-1.468643	-1.158394	-0.07785	-0.608581	0.0036035	-0.436167	0.7477308	-0.793981	-0.770407	1.047627	-1.066604	1.1069535	1.6601136	-0.279265	-0.419994	0.4325353	0.263
18	12	1.1032154	-0.040296	1.2673321	1.2890915	-0.735997	0.2880692	-0.586057	0.1893797	0.7823329	-0.267975	-0.450311	0.9367077	0.7083804	-0.468647	0.3545741	-0.246635	-0.009212	-0.595912	-0.575682	-0.1
19	13	-0.436905	0.9189662	0.9245908	-0.727219	0.9156787	-0.127867	0.7076416	0.0879624	-0.665271	-0.73798	0.3240978	0.2771921	0.2526243	-0.291896	-0.18452	1.1431737	-0.928709	0.6804696	0.0254365	-0.04
20	14	-5.401258	-5.450148	1.1863046	1.7362388	3.0491059	-1.763406	-1.559738	0.1608417	1.2330897	0.3451728	0.9172299	0.9701167	-0.266568	-0.47913	-0.526609	0.4720041	-0.725481	0.0750814	-0.406867	-2.19
21	15	1.492936	-1.029346	0.4547947	-1.438026	-1.555434	-0.720961	-1.080664	-0.053127	-1.978682	1.638076	1.0775424	-0.632047	-0.416957	0.0520105	-0.042979	-0.166432	0.3042414	0.5544325	0.0542295	-0.3
22	16	0.6948848	-1.361819	1.029221	0.8341593	-1.191209	1.3091088	-0.878586	0.4452901	-0.446196	0.5685207	1.0191506	1.2983287	0.4204803	-0.372651	-0.80798	-2.044557	0.5156635	0.6258473	-1.300408	-0.13
23	17	0.9624961	0.328461	-0.171479	2.1092041	1.1295656	1.6960377	0.1077116	0.5215022	-1.191311	0.7243963	1.6903299	0.4067736	-0.936421	0.9837394	0.7109108	-0.602232	0.4024844	-1.737162	-2.027612	-0.26
24	18	1.1666164	0.5021201	-0.0673	2.2615692	0.4288042	0.0894735	0.2411466	0.1380817	-0.989162	0.922175	0.7447858	-0.531377	-2.105346	1.1268701	0.0030753	0.4244245	-0.454475	-0.098871	-0.816597	-0.30
25	18	0.2474911	0.2776656	1.1854708	-0.092603	-1.314394	-0.150116	-0.946365	-1.617935	1.5440714	-0.829881	-0.5832	0.5249332	-0.453375	0.0813931	1.5552042	-1.396895	0.7831308	0.4366212	2.1778072	-0.23
26	22	-1.946525	-0.044901	-0.40557	-1.013057	2.9419677	2.9550534	-0.063063	0.8555463	0.0499669	0.5737425	-0.081257	-0.215745	0.0441606	0.0338978	1.1907177	0.5788435	-0.975667	0.0440628	0.4886029	-0.21
27	22	-2.074295	-0.121482	1.3220206	0.4100075	0.2951975	-0.959537	0.5439855	-0.104627	0.475664	0.1494506	-0.856566	-0.180523	-0.655233	-0.279797	-0.211668	-0.333321	0.0107511	-0.488473	0.505751	-0.38
28	23	1.1732846	0.3534979	0.2839051	1.1335633	-0.172577	-0.916054	0.3690248	-0.32726	-0.246651	-0.046139	-0.143419	0.9793504	1.4922854	0.1014175	0.7614775	-0.014584	-0.51164	-0.325056	-0.390934	0.027
29	23	1.3227073	-0.174041	0.434555	0.5760377	-0.836758	-0.831083	-0.264905	-0.220982	-1.071425	0.8685585	-0.641506	-0.111316	0.3614854	0.1719451	0.7821665	-1.355871	-0.216935	1.2717654	-1.240622	-0.52
30	23	-0.414289	0.9054373	1.7274529	1.4734713	0.0074427	-0.200331	0.7402283	-0.029247	-0.593392	-0.346188	-0.012142	0.7867963	0.6359539	-0.086324	0.0768037	-1.405919	0.7755917	-0.942889	0.5439695	0.097
31	23	1.0593871	-0.175319	1.2661296	1.18611	-0.786002	0.5784353	-0.767084	0.4010461	0.6994997	-0.064738	1.0482925	1.0056184	-0.542002	-0.039915	-0.218683	0.0044757	-0.193554	0.042388	-0.277834	-0.17

Figure 1: Screenshot of dataset

We conducted exploratory data analysis by checking the correlation between the features using the Pearson correlation matrix. By plotting the matrix with a heatmap that shows the correlation clearly, we saw from the graph that the features have really low correlation and are relatively independent with each other, meaning we could indeed use all features obtained from PCA. We

then extracted the 'Class' column from the dataset to establish a feature dataset and a target dataset. Afterwards we used holdout method, splitting the datasets into 70% training and 30% test sets. Because our data is highly unbalanced, we decided to use stratified sampling to make sure that the training and test data have the same distribution in the target variable as the original data. Random state was also used to ensure the same split every time. In addition, beforing running Complement Naive Bayes, we used a min-max scaler of range 0 to 1 on the feature datasets because Complement NB does not work with negative values.

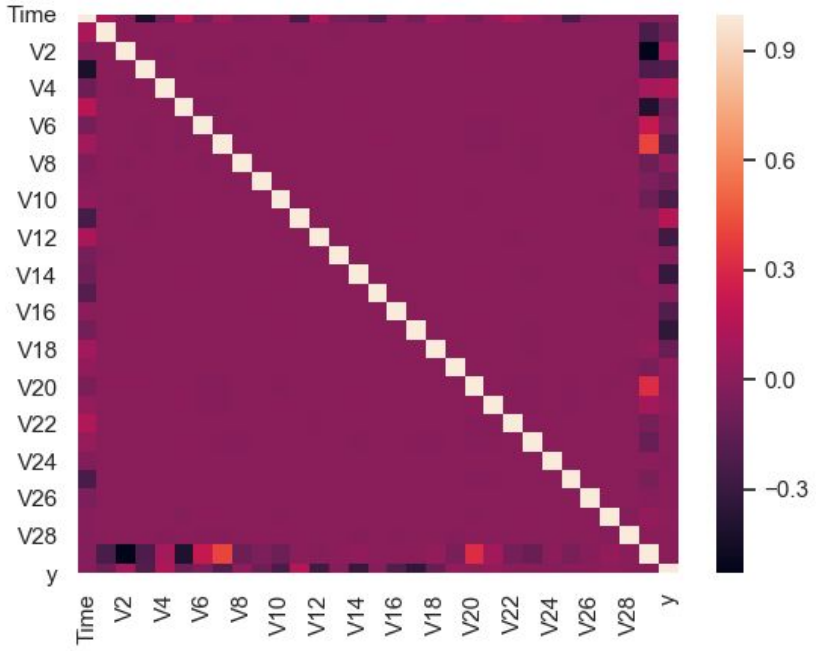


Figure 2: Heatmap of correlation

Finally, the models we selected were KNN, Decision Tree, Logistic Regression, Complement Naive Bayes, Voting, Bagging, Random Forest, XGBoost and MLP. Note that we did not include SVM because it took a really long time to converge, and it was hard for us to do further analysis. Also, our MLP model did not converge, as we could not find a parameter that would make it converge in a reasonable time. As a result, the scores obtained from MLP could not be considered entirely accurate, nor did they come off exactly impressive.

We started with the most basic model KNN and tuned the parameter 'n_neighbors' with GridSearchCV and use 'precision' as the scoring parameter. The optimal value we got was 3. Then we tuned parameters for Decision Tree, and the parameters we chose were 'max_depth' and 'min_samples_leaf". With GridSearchCV we were able to find the optimal parameters: max_depth = 7 and min_samples_leaf = 5. For Perceptron, we decided to set the parameter 'max_iter' to be 1000 so that the model would converge. For Logistic Regression, we used the LogisticRegressionCV and set scoring as 'precision'. For ComplementNB, we didn't tune any parameter.

As stated before, Logistic Regression, Decision Tree and Complement Naive Bayes were selected as base learners for Voting. The weight we gave to Decision Tree was 2 while the other two had weight 1 as Decision Tree has the best performance among all three. For Logistic Regression we assigned 'max_iter' to be 1000 to ensure convergence. Then using the optimal parameters that we obtained from tuning parameters of the Decision Tree classifier, along with Logistic Regression and Complement NB, we could run the Voting ensemble and get the improved results we desired. Also utilizing the optimal values from Decision Tree, we gave bagging another run and were delighted to see that the scores improved as well. For Random Forest, the parameters we tuned are 'max_depth', 'min_samples_leaf', 'max_features' and 'n_estimators'. Again, we used GridSearchCV and the optimal value we got were max_depth = 19, min_samples_leaf = 5, max_features = 3, and n_estimators = 30. Lastly, we tuned three parameters for XGBoost: 'max_depth', 'n_estimators' and 'learning_rate'. Running GridSearchCV yielded the values that were optimal for these parameters: max_depth = 6, n_estimators = 350 and learning_rate = 0.3.

Below is a table that shows how the models performed in terms of the four metrics that we decided on to compare these algorithms:

Model	Precision	Recall	AP	AUPRC
KNN	0.900	0.061	0.06	0.180
Decision Tree	0.848	0.716	0.61	0.746
Logistic Regression	0.833	0.507	0.42	0.707
Naive Bayes	0.706	0.730	0.52	0.639
Perceptron	0.000	0.000	0.00	0.003
Random Forest	0.940	0.720	0.68	0.813
Voting	0.913	0.709	0.65	0.794
Bagging	0.892	0.784	0.70	0.813

XGBoost	0.942	0.770	0.73	0.842
Neural Network*	0.678	0.784	0.60	0.695

Table 1: Metric scores of all models

To make the above outcomes easier to interpret, we conducted some visualization by plotting the precision-recall curves of these models in the same graph so that we could more conveniently and directly draw comparisons between the models' performances in order to judge which of them would be more suited for our task at hand. Remember that when one curve is placed above another, the above curve will have a larger area under the precision-recall curve (AUPRC) than the one below, and because AUPRC is one of the metrics employed by us to measure the performances of the models, the model corresponding to the curve that is above thus has a better performance than the model corresponding to the curve below. Here is what the graph looks like:

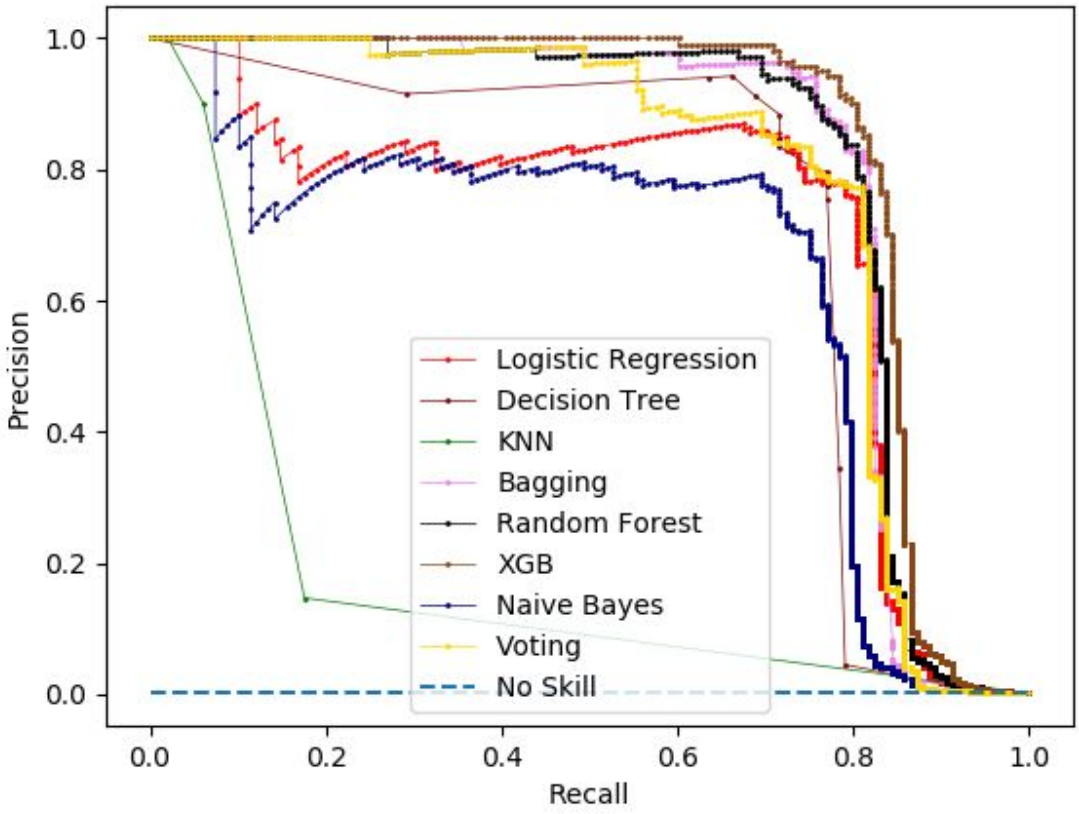


Figure 3: Precision-recall curves of models

Since we did not manage to get the MLP model to converge, we decided not to include it in the final graph. Perceptron was also left out of the group because its scores were fairly unsatisfactory

(to say the least). From the resulting graph above it is easy to see at first sight that one of the most used models nowadays XGBoost performed better than the others while the most fundamental model KNN did the poorest job of identifying credit card frauds. If we take a closer look at the remaining curves, it is not that difficult for us to see that both Random Forest and Bagging did a pretty good job, whereas Logistic Regression and Naive Bayes did not appear to be fit for the task. And in the middle we have Decision Tree and Voting. We were kind of surprised by how good Decision Tree did, considering it is a relatively basic algorithm.

Discussion

Note that precision score could not be regarded as persuasive enough for us to judge the performance of a model. A high precision score only means there are few false positives (FP). As mentioned before, all models had a high accuracy of prediction and lots of true negatives (TN). In this case, all models had few FPs. So we decided to rely on recall score, which gives the ratio of correct classifications of positive classes among all positive classes.

Among all the models we used for this project, KNN and Perceptron were the ones with the worst performance judging by our model assessment strategies. KNN has an average recall score of 0.06 while the recall score of Perceptron is almost zero, which means it failed to label any positive class correctly. We were curious about these results and believed that they were related to the logic and algorithm of these models. From our analysis, KNN simply categorizes data based on classes of nearest points. As our data is extremely unbalanced (0.172% of positive class), and we tuned the optimal 'n_neighbors' with value 3, it is really unlikely that for a positive class, there are more than one positive classes among the three nearest points to it. This fact also suggests that the minority class in our dataset does not concentrate in an area of subspace. The bad performance of Perceptron could also be explained with its underlying mechanics. Perceptron is a kind of linear model that looks for a linear hyperplane to do binary classification. As we gave both classes an equal weight, the penalty for misclassifying a positive class was not significant enough to reflect such difference. We believed that giving the positive class more weights, or apply more margin to the positive class would yield a better result. This was similar to the mechanism of SVM, but the time to train an SVM model for this project was intimidating.

So we decided to try random undersampling as suggested by the most popular attempt on Kaggle. We used RandomUnderSampler from the imbalanced-learn API. There was a significant improvement in performances for both models after our update of data.

Model	Precision	Recall	AP	AUPRC
KNN	0.613	0.642	0.57	0.665
Perceptron	0.5	1	0.5	0.469

Table 2: Result of KNN and Perceptron after Undersampling

The recall score and AUPRC of KNN and Perceptron increased greatly. Perceptron had a recall score of 1 for this new balanced dataset which means it correctly labelled all positive classes.

Logistic Regression and Complement Naive Bayes have a medium performance just better than KNN and Perceptron. Logistic Regression was a generalized linear model. Although it has a sigmoid loss function $\sigma(z) = \frac{1}{1+exp(-z)}$, z is still a linear function of input features. This nature would explain the normal performance of Logistic Regression as it is also imperfect to handle unbalanced dataset.

We could see the top five models all have tree structures: Decision Tree, Random Forest, Voting with Decision Tree having half of the weights, Bagging with Decision Tree as the basic learner, Random Forest and XGBoost. This gave us a basic idea that tree structures would be the priority for our project. They tackle our problem by learning a hierarchy of if/else questions and could force both classes to be addressed. Note that simple Decision Tree model or Voting did not perform much better than Logistic Regression and Naive Bayes. Ensemble methods like Bagging, Random Forest and XGBoost were evidently better than all of the rest models as Bagging has more estimators and Random Forest and XGBoost indeed employs randomness when finding a split in each node. They would be more likely to find and group the minority classes into an area of subspaces.

Also, with the data after random undersampling, our MLP model managed to converge. Its performance is better than simple Perceptron on the same data.

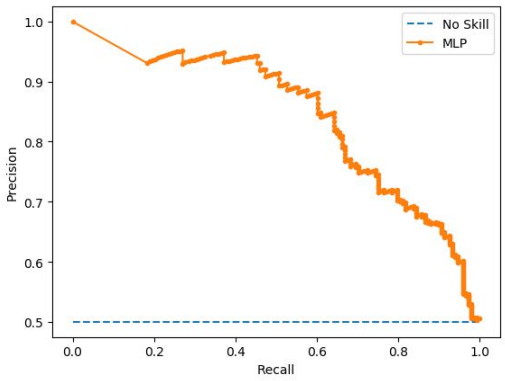


Figure 4: Precision-recall curve of MLP

Model	P	R	AP	AUPRC
MLP	0.881	0.601	0.73	0.843

Table 3: Metric Scores of MLP

Future Improvement

While we did succeed in utilizing undersampling as a useful method to handle our unbalanced data and getting noticeable improvement in some models that previously did not perform very well, we are aware that there is still room for development through future research.

First, there are still a number of things we might be able to experiment with to improve upon the models we chose. To be more specific, we could try to refine linear models that operate on large margins like SVM so that we would afterwards have a better idea of the effectiveness of such models on dealing with our task of credit card fraud identification. We did not use SVM in our present research because running the model was too time-consuming; nevertheless, we could definitely could work on this algorithm in the future to see if it would have some value in helping us achieve our research goal.

Second, we could explore more in the ensemble category. During our research we chose such ensemble methods as bagging, Random Forest and XGBoost, and we also used voting which combined several base learners for the meta learner. However, there is much more to dig deep into. Methods like stacking and blending may very well be just as useful as some of the best models we selected and even more.

Last but not least, we recognized it is very important that our approach could be very well generalized, so another possible future task is to assess the generalizability of our strategies and algorithms. It would be ideal if our approach to identifying credit card frauds can be applied to other real-life cases with similarly unbalanced data; on the other hand, we might need to rethink our methodology if it cannot be used to handle other unbalanced data effectively.

Conclusion

We draw the conclusion that among all the models we discussed, Random Forest, Bagging with Decision Tree as the base learner and XGBoost are the best models for this project. The reason stemmed from their tree structures as well as their number of estimators and use of randomness when finding a split in each node. We believed that for other extremely unbalanced datasets, people should first consider these three models to deal with this problem. KNN and Perceptron did poorly for this project. They would be more efficient and accurate when the data is balanced. Logistic Regression and Naive Bayes had a medium performance. There is certainly much more we can do in the future, in particular regarding SVM and Neural Network.

Contributions

We contributed equally on this project. We did the preprocessing of dataset together and split an even work distribution when tuning parameters, running all the models and recording the results.

Section	Contributor
Abstract	Yinfeng Wang
Introduction	Zirui Deng
Background	Together
Methods	Together
Experiments/Results	Together
Discussion	Together
Future Improvement	Zirui Deng
Conclusion	Yinfeng Wang

Code

Link to our group Google Drive folder:
https://drive.google.com/drive/folders/1T06JpstXWo0Bp7mjQhbi-Up7-Bjo7hMX

Reference

[1] Credit Card Fraud Detection. https://www.kaggle.com/mlg-ulb/creditcardfraud. Accessed: 2019-12-13.