















1. Introduction

GitHub Link: <https://github.com/kas5852/DataScienceStuff/kaggle>
(<https://github.com/kas5852/DataScienceStuff/kaggle>)

The goal of this lab report is to walk the reader through the steps I took in the Spring 2019 DSL Kaggle Competition. The lab report will go into detail regarding the different methods that I used throughout the lab as well as testing and validation techniques that I used. All the code used during the course of the project is available on the github link provided as well. Commit history can be viewed to see different milestones, but those milestones will also be discussed in this lab report.

1.1 Public and Private Score

Public Score

Public Leaderboard Private Leaderboard						
This leaderboard is calculated with approximately 50% of the test data. The final results will be based on the other 50%, so the final standings may be different. Raw Data Refresh						
#	Team Name	Kernel	Team Members	Score ?	Entries	Last
1	Allison Crow			0.91709	38	2d
2	Dhruv Sandesara			0.91378	27	2d
3	Ayush Srivastava			0.91096	21	2d
4	<u>KarimSabar</u>			0.91051	35	2d
5	Guilherme Zamorano			0.90985	37	2d
6	Qian Wen			0.90978	33	2d
7	Hassan Chughtai			0.90944	25	2d
8	Srinjoy Majumdar			0.90913	33	2d
9	Kevin Sheu			0.90911	32	2d
10	David Zehden			0.90746	21	2d
11	Henok Tadesse			0.90740	27	2d
12	Alexander Phillips			0.90714	35	2d
13	Kayvon Khosrowpour			0.90703	39	2d
14	Josh Luo			0.90601	32	2d

Private Score











Public Leaderboard

Private Leaderboard

The private leaderboard is calculated with approximately 50% of the test data.

This competition has completed. This leaderboard reflects the final standings.

 Refresh

#	Δpub	Team Name	Kernel	Team Members	Score ?	Entries	Last
1	▲1	Dhruv Sandesara			0.93617	27	2d
2	▲3	Guilherme Zamorano			0.93475	37	2d
3	▲1	<u>KarimSabar</u>			0.93400	35	2d
4	▼1	Ayush Srivastava			0.93374	21	2d
5	▲3	Srinjoy Majumdar			0.93321	33	2d
6	▲3	Kevin Sheu			0.93317	32	2d
7	▼1	Qian Wen			0.93192	33	2d
8	▼1	Hassan Chughtai			0.93125	25	2d
9	▲7	Elvin Hung			0.93120	26	2d
10	▲1	Henok Tadesse			0.93036	27	2d

1.2 Hardware

Desktop: (primarily used) Ryzen 1700X @3.9Ghz, 16GB DDR4 3133Mhz, Nvidia GTX 1070ti 8GB

Laptop: i7-6700HQ, 16GB DDR4, Nvidia GTX960M 2GB

1.3 Software

Editor: Jupyter Notebook

2. Choosing a model

Probably the hardest part of starting was choosing a model. Having never worked with any of the tools aside from the labs in class, I was extremely unfamiliar with all the different available models. Below I will discuss the different models I tried to use and the results they provided.

2.1 Regressor vs Classifier

In a nutshell, a regressor is used when you need to predict continuous values (such as house prices) and a classifier is used when predicting labels (such as marking an email spam or not spam). In the case of our Kaggle competition we were predicting labels (0 or 1) so a classifier was the natural choice.

2.2 Logistic Regression

This is the very first model I tried out. Despite the "regression" in its name, it is a classification model. This was mostly used to just get an idea of what kind of data I was dealing with, figure out the features and plot some graphs to get an idea of what I was doing. I never even got a chance to truly test it out since it was mentioned in class that it would produce a result of .5. Considering submissions were valuable, I didn't think it was a good idea to submit a logistic regression prediction. I decided to do research on other models I could use.

2.3 XGBoost

This is the model I primarily used and ended up sticking with. XGBoost is an ensemble model that utilizes boosting. It essentially combines different weak learners to create a stronger learner using gradient boosting and pruning. The default XGBClassifier without any modifications gave me an auc score of .86 on Kaggle. A big improvement on the base logistic regression model. However after doing some research I realized that the major strength of XGBoost lied in its hyper parameter tuning. With good hyper parameter tuning, XGBoost proved to be an extremely powerful tool that keeps on getting better if given enough time to find the best parameters using something like GridSearchCV from sklearn.

The reason I chose XGBoost over the second most popular classifier, RandomForest, was due to its multithreaded performance as well as extensive tutorials online. Due to the time constraints of the competition the model wasn't the only thing that had to learn quickly, I did as well. There were a number of amazing tutorials for XGBoost tuning and it made the most sense. This coupled with the fact that XGBoost was multithreaded and used all my cores made it the best choice. Other options were also considered, but overall XGBoost provided the most ease of use, best performance and ease of tuning.

2.4 Stacking

After I hit a wall with XGBoost hyper parameter tuning and was seeing little or no difference using GridSearchCV I decided that I need to use other methods. The first thing I came across was stacking. Stacking basically means to make predictions on a training set using different models and then finally using a meta classifier to use the results of those different models as features. In theory, it seemed like a great idea. There was also an extremely easy to use library called mlextrnd that allowed you to pass it different classifiers to stack. However, the results were suboptimal to say the least. My scores were hitting the mid .7s which was much lower than the .89 I'd get with XGBoost alone. I tried stacking different models hoping for better results. The first was KNearestNeighbors, LogisticRegression & RandomForest (which is an ensemble model itself). This gave me a very low score. Next I

tried adding my finely tuned XGBoost model into the mix. However it wasn't until multiple tests later that I realized that I'd have to find completely new hyperparameters if I wanted reach a decent score. This caused multiple issues since I'd have to essentially start fresh which would be too time consuming. Not to mention I had no idea how the different models work together, I was basically trying random things. I decided that if granted more time stacking may have been the better option, but considering how late it was into the competition I

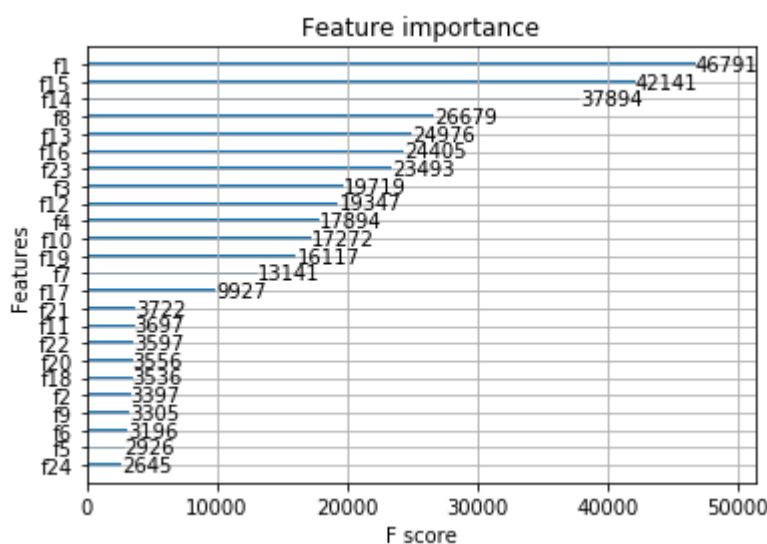
3. Feature Engineering

Knowing nothing about what the dataset represented definitely hindered progress. In the beginning I had believed it was impossible to feature engineer and I had to work with everything I was given. However, this proved to be untrue and I realized that there were two different ways that I could feature engineer.

3.1 Removing unimportant features

Reading through the XGBoost documentation I realized that there was a "plot_importance" function that allowed you to see the importance of different features on the model. Looking at the data myself there was no way to figure out what was important and what wasn't, so this function made everything a lot easier. There were 10 features ['f2', 'f5', 'f6', 'f9', 'f11', 'f18', 'f20', 'f21', 'f22', 'f24'] that seemed to have little to no impact on the model. After removing these features the results were interesting to say the least.

When doing 10 K Fold validation, the variance between my different folds went down significantly. The variance was initially around 1.5-1.6% but after taking out these features it went down to 1.3%. My overall KFold score went down (without tuning) by .02 but after tuning with the new features, the difference with KFold validation was minimal. However, the real difference was with Kaggle submissions. I had closed the gap between my feature engineered vs non feature engineered KFold score but my Kaggle score went down by a lot, even though the KFold scores were very similar. This could be due to the bias variance trade off that is discussed in the testing section but was very interesting to observe.



source my python notebook

3.2 Using Z scores to find outliers

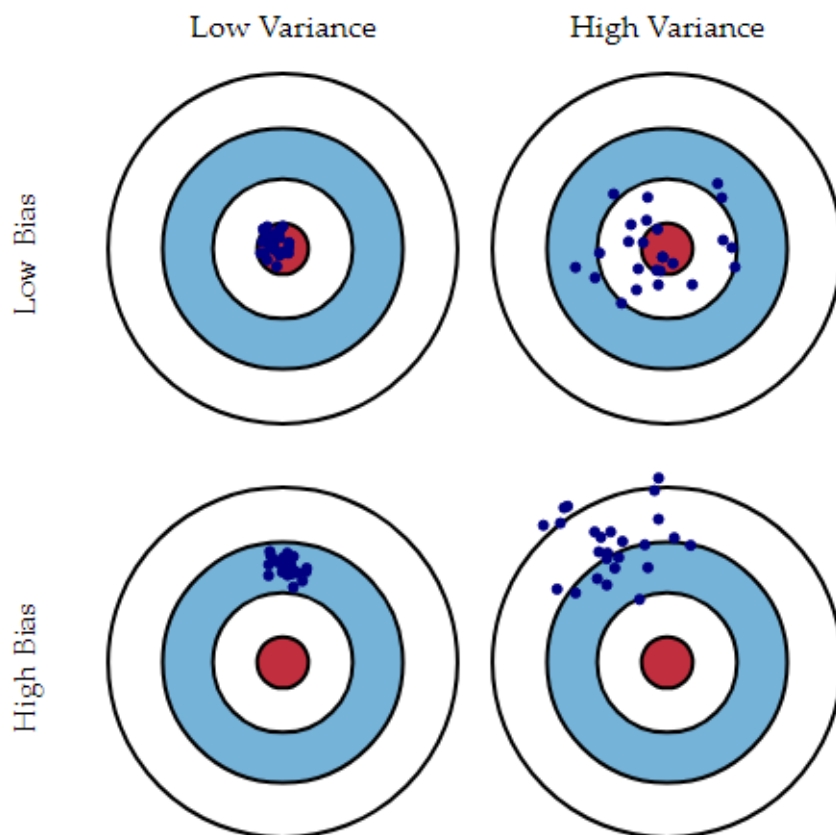
This was a method I briefly used and may have provided results, but seemed like it would require a lot of changes to parameters to make work. I realized that Z scores could have been used to find outliers to reduce the variance, but this again fit into the bias variance trade off. Overall, this technique should have been explored more and the next time around I will definitely start with this when feature engineering since just taking away features did not produce good results.

4. Hyperparameter Tuning

The biggest strength of XGBoost seemed to be its ability to be finely tune its hyperparamaters. The difference between tuning and not tuning yielded a difference of around +.5 to the auc score. However, this was after days using GridSearchCV and trying out different ranges of paramaters. This section of the report will go in detail about the different techniques I used for hyperparameter optimization as well as there pros and cons.

4.1 Bias Variance Trade-Off

This was an issue I dealt with the entire time I was tuning my hyper parameters. I have included a diagram below to show the impact it has, but due to excessive tuning I was running into many issues. I'd either have a high bias and low variance, which gave me bad Kaggle scores, or low bias and high variance which made my Kaggle submissions somewhat unpredictable. After many days of trial and error it seemed like a variance of 1.5% seemed to be okay. Anything lower and my score would never go up and anything higher and my KFold would be too inaccurate causing me to lose submissions. It was also important to look at variance specifically since the only way thing I could measure without submitting. To figure out the bias I'd have to do multiple Kaggle submissions which was not a viable option. Variance, however, was something I had control of. I could significantly reduce the variance by taking out features, but as discussed above in the feature engineering section that caused a lower Kaggle score. I hope to learn more ways to mitigate the effects I felt of overtuning and it's something I plan to improve on during the next competition with more research.



source <http://scott.fortmann-roe.com/docs/BiasVariance.html>

4.2 Bayesian Optimization and Hyperopt

A technique I briefly tried was Bayesian optimization. The idea behind it is that instead of set ranges and trying every permutation by brute force, it chooses random pairs and provides the best out of them. This was much quicker than grid search for validation as well as finding good parameters. However in hindsight I was using it too early on. I tried this technique once I started hitting what seemed like a brick wall with grid search. But, my bounds were still far too loose for Bayesian optimization. By having tighter bounds and passing them to hyperopt (which is a module used to implement bayesian optimization) I would have been able to find much better values than with grid search. The reason for this is because towards the end of the competition, values kept being repeated and the plug and chug method wasn't effective anymore, I needed to tune them collectively. But, with grid search that still wasn't possible. This would have been the best time to use bayesian optimization. But using it in the beginning like I did gave very bad results

4.3 GridSearchCV

GridSearchCV is contained within the sklearn module and used for finding good hyper parameters. It essentially allows you to pass it a set of parameters as well as a classifier to test all different permutations with the parameters you specify. It then runs 5 fold cross validation by default to give you the best parameters out of the bunch. Naturally, since it is essentially a brute force algorithm it takes a long time. But I found that it was much quicker to run parameters as individuals and then combine them at the end. This can be seen in my code where I have different "paramtests" for the different parameters. This was not the optimal way since you'd eventually hit a brick wall (like I did at many points) since you'd essentially be plugging in the same things again and again. To write out my process it would be:

1) Find optimal max_depth and min_child_weight with current parameters

- If close to the lower bound or upper bound, adjust bounds and re run

- Keep doing until the optimal is between two bounds for both min_child_weight and max_depth

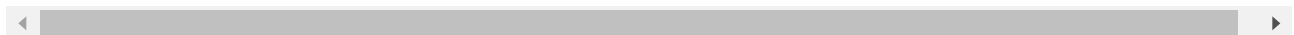
2) Find optimal gamma with new max_depth and new min_child_weight plugged in

- If close to upper or lower bound, adjust and do until value is between both bounds



3) Find optimal subsample and cosubsample_bytree with new gamma, max_depth and min_child_weight

- If close to upper or lower bound, adjust and do until value is between both bounds



4) Find optimal reg_alpha and reg_lambda with new values found above

- If close to upper or lower bound, adjust and do until value is between both bounds



5) Re-find gamma with new bounds

```

In [101]: param_test1 = {
    'max_depth':range(15,21,1),
    'min_child_weight':range(0,3,1)
}
gsearch1 = GridSearchCV(estimator = XGBClassifier(
    learning_rate =0.09,
    n_estimators=300,
    max_depth=32,
    min_child_weight=0,
    gamma=0.0,
    subsample=0.8,
    colsample_bytree=0.15,
    objective= 'binary:logistic',
    nthread=16,
    scale_pos_weight=1,
    seed=42,
    reg_lambda=0
),
    param_grid = param_test1, scoring='roc_auc',n_jobs=-1,iid=False, cv=10)
gsearch1.fit(dataset[predictors],dataset['Y'])
gsearch1.cv_results_, gsearch1.best_params_, gsearch1.best_score_

'split8_train_score': array([1.          , 1.          , 0.99999663, 1.          , 1.          ,
    0.99999781, 1.          , 1.          , 0.9999984 , 1.          ,
    1.          , 0.99999924, 1.          , 1.          , 0.99999933,
    1.          , 1.          , 0.99999958]),
'split9_train_score': array([1.          , 1.          , 0.99999857, 1.          , 1.          ,
    0.99999309, 1.          , 1.          , 0.9999952 , 1.          ,
    1.          , 0.99999916, 1.          , 1.          , 0.99999907,
    1.          , 1.          , 0.99999848]),
'mean_train_score': array([1.          , 1.          , 0.99999692, 1.          , 1.          ,
    0.99999734, 1.          , 1.          , 0.99999844, 1.          ,
    1.          , 0.99999904, 1.          , 1.          , 0.99999913,
    1.          , 1.          , 0.99999938]),
'std_train_score': array([0.00000000e+00, 0.00000000e+00, 1.63963725e-06, 0.00000000e+00,
    0.00000000e+00, 1.94480249e-06, 0.00000000e+00, 0.00000000e+00,
    1.33038170e-06, 0.00000000e+00, 0.00000000e+00, 7.40778037e-07,
    0.00000000e+00, 0.00000000e+00, 6.88758320e-07, 0.00000000e+00,
    0.00000000e+00, 5.14694297e-07]),
{'max_depth': 20, 'min_child_weight': 1},
0.9047214132923713)

```

An example of a paramtest for max_depth and min_child_weight

t

Eventually, I had to change the 5 fold cross validation to 10 fold cross validation to get more accurate results as my bounds were getting tighter and tighter. However, I increased the learning rate to .1 and decreased n_estimators to 140 to make learning quicker. The goal was to reach the local minima of error eventually, but that would require a much longer and more accurate grid search. Due to the amount of time it took to do grid search, I explored other options such as Bayesian optimization using hyperopt.

4.3.1 Tuning min_child_weight and max_depth

min_child_weight and max_depth seemed to have some of the most significant impacts on the overall auc score. As the names imply, the max_depth determines the max depth a tree can go while boosting before being pruned and min_child_weight determines what the minimum weight a child can have in a tree. For a long time I was afraid that a larger max_depth would cause overfitting. However after doing some research and using 10 Kfold validation I was confident that as long as my scores were going up I didn't need to bound myself. I had initially limited myself to a max_depth of 24 since that is how many features there were. However, going all the way up to 29 is what helped me break the .91 auc score on the public leaderboard. Another place I was putting artificial limits on myself was a min_child_weight of 1. I refused to go under 1 (even though my grid search suggested I should) since I was again afraid of over fitting. But again, setting a min_child_weight of 0 really helped me make some real progress.

4.3.2 Tuning subsample and cosubsample_bytree

Due to these values regularly being under 1 it was a little bit harder to tune them. I had to use small ranges (such as .78-.85 for subsample and .3-.4 for cosample) during my grid searches. But, even this took a long time since there were many permutations. The high learning rate and low `n_estimators` helped but once I was doing 10 fold validation on everything it was taking a very long time. But, it was needed considering I was hitting the upper bounds of tuning parameters one by one. Ideally, I'd want to tune them all together and leave it going for a couple of days. Another option that I wanted to explore was using bayesian optimization now that my ranges were smaller. But, again, due to time constraints I was not sure whether it was worth trying out towards the end.

4.3.3 Tuning regularization paramaters, gamma, reg_alpha, reg_lambda

Since these were all regularization paramaters I believed that it was best to tune them together. So, I would turn them to 0 when tuning everything else and then I would tune these at the end. It seemed to work best that way since it gave me an idea of what kind of regularization I needed to do. I realized that whenever I had a value for `gamma > 0` my `reg_alpha` and `reg_lambda` would go down. But, Kaggle seemed to not like the `gamma` for some reason so I tried to focus more on tuning `reg_alpha` and `reg_lambda`.

4.3.4 Tuning n_estimators and learning rate

The trade off to using low `n_estimators` and a high learning rate while doing grid searches resulted in inaccurate cross validation scores during the grid search. This means I'd have to tune all parameters and then test them all together using a learning rate of .01. I chose .01 since it seemed to work the best initially. Once I had all my parameters I would slightly lower the learning rate even more to .009. If the score went up, I'd keep tuning it down to get the optimal rate. If the score went down, I'd increase it a bit and see what happens then adjust accordingly.

`n_estimators` was a bit more involved then that. I would start with 5000 and then use a function called `modelfit` to use an `early_stopping` rate of 50 to find the best `n_estimators`, this would also use 10 fold validation. I eventually had to increase my `early stopping` rate to 200 which increased the time it took to run by a lot, but was worth it in the end of considerable gains to my `auc` score.

```
: def modelfit(alg, dtrain, predictors, useTrainCV=True, cv_folds=5, early_stopping_rounds=50):

    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(dtrain[predictors].values, label=dtrain[target].values)
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'], nfold=cv_folds,
            metrics='auc', early_stopping_rounds=early_stopping_rounds)
        alg.set_params(n_estimators=cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain['Y'], eval_metric='auc')
    print("Model Fitted")
```

Modelfit function I utilized

5. Testing and Validation

5.1 Submission Methodology

Considering submissions were our most valuable resource, figuring out how to use them tactically was essential. In the beginning of the competition submissions were my only form of validation. This worked for the first day since big improvements were easy to make. But, as my model started to become more fine tuned and changes become more incremental, submitting to test was a terrible option. Which is why it had become extremely important to find a good way to test before I submit. The idea was to submit only if there was a significant increase in my score, but for that I'd need to have an accurate testing method. There were two methods I utilized, `test_train_split` and 10 Kfold cross validation from sklearn. The metric used to measure improvement was `roc_auc`.

5.1.1 test_train_split

Due to the small amount of submissions available a day, it was crucial to find a way to accurately measure progress. The first option that I started using was the `test_train_split` function provided by sklearn. Providing 77% of the data for training and 33% for testing and validation. Although in theory it seemed like a good idea, it ended up causing over fitting. By using the same testing and training set over and over again the model had started to overfit. Another problem was that the auc and accuracy scores provided by `test_train_split` were not accurate. I could get a 90% on my test set but end up in the mid 80s (if not lower) on Kaggle submissions. Considering how close the competition was such inaccuracy was not acceptable.

5.1.2 10 Kfold Cross-Validation

This is when I decided to use 10 Kfold validation. Kfold validation essentially creates random splits in your data for testing and training. For example, if you tell Kfold to do 10 splits, it will train on 9 "folds" and then test on 1 fold of data. It will do this for every split. There were multiple advantages to using KFold over `test_train_split` and testing myself. First off, it was much more accurate and representative of the actual scores I would receive on Kaggle, if not more conservative. This allowed me to use my submissions alot more tactically than any other method of validation. Kfold validation also essentially took away any fears of overfitting. Due to the nature of the data splits and the constant testing, it would be extremely difficult to overfit to the training data. This was further proved by score improvements on Kaggle which was also essentially impossible to overfit too. However, it should be noted that these advantages are concerning 10 Kfold validation only. With a lower amount of folds (such as 5) I noticed there was a sharp decrease in accuracy. 10 Kfold also took a very long time (around 7 minutes on my desktop) with a low learning rate.

One interesting observation I made was Kfold reactions to feature engineering. After using the `plot_importance` function and realizing 10 of the features were significantly less important than the others, I removed them. Kfold provided better results and less variance (this is discussed further in the bias variance trade off section) but my Kaggle score would go down (even after paramater tuning). Also, once I hit an auc .91 on Kaggle 10 Kfold started being less accurate and less conservative. Under .9% 10 Kfold would provide me a lower auc than Kaggle would. But, over .9% would provide me a higher score auc than Kaggle does. This was an issue I was not able to overcome since it was discovered so late in the competition. One guess I would have would be the fact that the training data was skewed towards '1' and the test set could have been a bit more balanced. This would explain why the model did better on test splits then it did on the actual test set.

6. Conclusion

One of the most important things I learned from this lab was the value of testing and accuracy. By having a way to accurately measure my score without submitting my progress skyrocketed. By keeping things consistent and having a plan and metrics to test with, it is much easier to know where to go. In the future, I'd like to experiment more with stacking since I believe it will provide better overall results, even if it will be harder to implement. Tools such as bayeseian optimization are something I'd also like to utilize extensively in the future as well as more feature engineering.