MIS382N Fall 2019 Kaggle competition

https://www.kaggle.com/c/mis382n-fall-2019

- Abhilash Vikram Gupta

This will describe our day-to-day analyses and evolving strategies to get the highest possible ROC-AUC score on the data.

Ideally a data science exercise starts with a lot of exploratory analysis, has some breakthroughs, constructs features, implements algorithms, refines some and ultimately converges on a final solution. The 5-submissions-a-day format of this competition makes it infeasible to proceed in this manner. This report (which doubles as a journal) will describe a more iterative approach in which we will attempt a handful of techniques each day, see how they perform and use the feedback to deduce what works and what does not.

It is observed that this competition is clocked at UTC time. It makes sense to make 5 submissions each day ending 7 PM, and be sure to reflect upon what went right/wrong in some of the remaining time.

Day 1

Ideas

- We have precious little time here, if we want to beat the 7:00 PM day deadline.
- First glance: supervised binary classification with a heavy bias. Some features look like categorical variables, but our belief is that this is deceptive and that they are in fact count-like. This is because not only are the variables missing the 0 key but also are missing some other values in between. Some of the other columns show heavy variation and others still are quite large (5-digit).
- Key concepts to use:
    - Recursive Feature Selection: with cross validation to select the best features.
    - StratifiedKFold: Stratified because we have an imbalanced dataset with a lot of variance between features.

```
def get_best_features(model, data, step=1):
    rfecv = RFECV(estimator=model, step=step, cv=StratifiedKFold(5),
scoring='roc_auc', n_jobs=-1)
    rfecv.fit(data, y)

    return get_ind(rfecv.ranking_ == 1)
```

- Scaling followed by logistic regression gives a bad ROC-AUC score and is hopelessly biased towards the dominant label.
- A plain decision tree overfits hard. In fact, it gives an AUC of 1.0 on train data using just `f14`.
- RFC does alright, but the score does not rise above 0.85. If we have aspirations of the highest possible score, we need to use gradient boosting.
- Considering LightGBM and XGBoost right now -- start with LightGBM because we have never used it and XGBoost is slow.

- Tuning LightGBM -- key realizations:
    - LGBM attempts to replicate scikit-learn's API.
    - The `goss` `boosting_type` parameter makes LGBM boosting *insanely* fast.
    - We need to set class weights somewhere.
    - Use all available processors, always.
    - Focus heavily on controlling tree depth and learning depth (eta). There appear to be maxima -- combinations of eta, depth and the number of estimators that give good results at certain spots and bad results at others.

```
for feats in feats_to_try:

    grid = GridSearchCV(xgb.XGBClassifier(),
                        tuning_parameters,
                        cv=StratifiedKFold(5),
                        scoring='roc_auc',
                        n_jobs=-1)

    grid.fit(X[:, feats], y)

    print(f'Best parameters {grid.best_params_}.')
    print(f'Best auc score is {grid.best_score_}.')
```

Day's best: 0.87978
Overall Best: 0.87978

Reflections

What went well

1. Got five submissions off despite short notice.
2. We seem to have the top spot, for now.

What we learnt

1. (Maybe) XGBoost might be more accurate than LightGBM but takes longer on larger datasets. One smart thing to do would be to use LightGBM on early on, when doing iterative testing, and use equivalent settings with XGBoost as the end of the competition draws near.
2. It is difficult to say which are the true features. It might be right to vary the features a little, taking 7 to 10 at a time and see how that fares.

How we can improve

1. Over-relying on LightGBM might leave us blind to other ways of solving the problem.
2. Need to learn more about decision trees and gradient boosting to tune better.
3. Could possibly use neural networks? Can investigate.

Day 2

Ideas

- There seems to be a bit of an accuracy jump with `f1`, `f4`, `f8`, `f13`, `f14`, `f15`, `f16` and `f18`

that subsides when we add `f7` to the mix. This was as an observation from day 1 that we failed to record.
- Spend the day attempting neural networks. Know that we cannot use the convolutional neural networks based predictions that we did in image classification. Have to try something else.
- Procured AUC scores of around 0.5 everytime. Spent some time fiddling, but then dropped it.

Day's best: ~0.45 (Obviously, something went very wrong. Still don't quite know what. Did not have time to investigate.)
Overall Best: 0.87978

Reflections

What went well

1. Very little. Using deep learning may be possible, but it will require time and patience.

What we learnt

1. Neural networks are terrible for classification with imbalanced datasets (at least, straight out of the box).
2. Focus on gradient boosting for now. Can come back to neural networks later.

How we can improve

1. Learning and using XGBoost is something that should definitely be done.
2. Spend more time with keras, etc. but later.

Day 3

Ideas

- This is XGBoost day. Attempt to master the different parameters being used and beat the score set by LightGBM.
- It looks like XGBoost also supports the familiar scikit-learn format api too. But we will use the XGBoost native format this time.

```
param = {'max_depth': 6,
         'eta': 0.1,
         'objective':'binary:logistic',
         'n_estimators': 200,
         'booster': 'gbtree',
         'tree_method': 'auto',
         'reg_alpha': 0.001,
         'reg_lambda': 0.9,
         'eval_metric': 'auc',
         'nthread': -1}

num_round = 120

# set scale_post_weight each time
def fpreproc(dtrain, dtest, param):
    label = dtrain.get_label()
    ratio = float(np.sum(label == 0)) / np.sum(label == 1)
    param['scale_pos_weight'] = ratio
    return (dtrain, dtest, param)

result = xgb.cv(param,
                dtrain,
                num_round,
                nfold=5,
                stratified=True,
                metrics={ 'auc' },
                fpreproc=fpreproc)
```

Day's best: 0.88192
Overall best: 0.88192

Reflections

What went well

1. A miniscule improvement in the best prediction set.
2. A better understanding of gradient boosting and XGBoost.

What we learnt

1. XGBoost is far, far slower than LightGBM. This means more time waiting for results and less time actually looking at them.
2. Setting the correct `scale_pos_weight` (number of negative class divided by number of positive class) yields an increase of ~0.002.
3. The increased AUC ROC score today may, in my opinion, be attributed to a combination of randomness and a better understanding of l1 and l2 regularization -- little else.
4. Unless there are other as of yet unknown benefits of XGBoost, we should prefer LightGBM in the long term.

How we can improve

1. Do some data engineering. Attempt polynomial features and interaction terms.

Day 4

Ideas

- Spend the day on feature creation and decomposition.
- Attempt an auto-tuning process. Each loop should:
    - Find the most important features in the original dataset for the best current model.
    - Create new features that are exponential (eg. **0.5, **2) of these original features.
    - Scale the same original features and create interaction terms (added and subtracted).
    - Use recursive feature elimination with cross validation to select the best of all of the features combined.
    - Take the best parameters in the previously selected model. Create the parameters for a grid search by slightly altering the parameters.
    - Weigh the variation in parameters by multiplying the variation with a decomposition term -- a number that keeps decreasing until it hits zero.
    - Use the generated range of parameters to select the best model using a grid search with cross validation.
    - Decrement the decomposition term and repeat the process.

```
def generate_train(dataframe, model):
    train_dataframe = dataframe.drop('Y', axis='columns')
    labels = dataframe['Y']

    feature_ratings = model.fit(train_dataframe.values,
labels).feature_importances_
    first, second, third = get_best_ind(feature_ratings, 3, 1)[0]
    first_name, second_name, third_name = indices_to_names(train_dataframe,
[first, second, third])

    train_dataframe[first_name + '*1.5'] = train_dataframe[first_name] ** 1.5
    train_dataframe[first_name + '*2'] = train_dataframe[first_name] ** 2

    relevant_columns = train_dataframe[[first_name, second_name,
third_name]].values
    scaled = StandardScaler().fit_transform(relevant_columns)
    fused = PCA(n_components=1).fit_transform(scaled[:, 1:]).ravel()

    train_dataframe[first_name + '+fused'] = scaled[:, 0] + fused
    train_dataframe[first_name + '-fused'] = scaled[:, 0] - fused

    return train_dataframe

def alter_params(params, dec):
    new_params = {}
    new_params['n_estimators'] = [round(params['n_estimators'] + np.ceil(dec)),
                                  round(params['n_estimators'] - np.ceil(dec))]
    new_params['learning_rate'] = [params['learning_rate'] + 0.001 * dec,
                                   params['learning_rate'] - 0.001 * dec]
    new_params['n_estimators'] = [round(params['num_leaves'] + np.ceil(0.05 *
dec)),
                                  round(params['num_leaves'] - np.ceil(0.05 *
dec))]
    new_params['reg_alpha'] = [params['reg_alpha'] * (1 + (0.02 * dec)),
                               params['reg_alpha'] * (1 - (0.02 * dec))]
    new_params['reg_lambda'] = [params['reg_lambda'] + (0.05 * dec),
                                params['reg_lambda'] - (0.05 * dec)]
    return new_params

def tune_iteratively(params, dec):
    if dec < 1:
        return

    print(f'Variation decay factor {dec}.')

    print('Feature tuning...')
    new_train = generate_train(df, model)

    old_model = xgb.XGBClassifier(**const_params, **params)
    new_feats = ind_to_name(new_train, get_best_features(old_model,
new_train.values))

    variable_params = alter_params(params, dec)
    print(variable_params)

    print('Param tuning...')
    grid = GridSearchCV(xgb.XGBClassifier(**const_params),
                        variable_params,
                        cv=StratifiedKFold(5),
                        scoring='roc_auc',
                        n_jobs=-1)

    grid.fit(new_train[new_feats].values, y)

    with open('results.csv','a') as f:
        f.write('\n, '.join([str(dec), str(grid.best_score_),
str(grid.best_params_), str(new_feats)]))

    tune_iteratively(grid.best_params_, dec - 1)
```

Day's best: null (Did not, in the end, submit any scores; had other things to attend to.)
Overall best: 0.88192

Reflections

What went well

1. Implemented the proposed auto-tuner. We now know that the design is possible and works, albeit inefficiently.
2. Experimented with some feature engineering.

What we learnt

1. The tool that we devised is inefficient, but automated. It could be used with success when given a dataset that we are not familiar with, or when one has other tasks. Focused human effort, however, yields better results.
2. Feature manipulation has almost no positive effect on our classifier. In fact, it even yields slightly lower scores.

How we can improve

1. Attempt to make five submissions a day (not submitting is quite bad).

Day 5

Ideas

- We noticed earlier that for practical purposes, LightBGM with `goss` proved to be as accurate and much faster than XGBoost. Spend some hours tuning it.

```
selected_cols = ['f14', 'f1', 'f15', 'f16', 'f8', 'f4', 'f13', 'f19', 'f17']

fix_param = {
    'boosting_type':                 'goss',
    'metric':                        'auc',
    'objective':                     'binary',
    'scale_pos_weight':              scale_pos_weight,
    'n_jobs':                        -1
}

var_param = {
      'n_estimators':                  [933, 934],
      'learning_rate':                 [0.097474],
      'min_child_weight':              [1e-3, 1e-9],
      'min_split_gain':                [6e-6],
      'colsample_bytree':              [0.4],
      'reg_alpha':                     [0.05],
      'reg_lambda':                    [0.89995]
}
```

Day's best: 0.89364
Overall best: 0.89364

Reflections

What went well

1. Had the first significant improvement over our initial auc score. The overall improvement is just ~0.014, but that could prove significant.

What we learnt

1. Using the model's `feature_importances_` attribute does not always yield the best features. This includes using feature selection methods that rely on the attribute, such

as recursive feature elimination RFE.
2. We have yet to explore all the hyperparameters of gradient boosting. Learning more about them would be wise.

How we can improve

1. Study data, use intuition and handpick to synthesize and/or obtain the best possible features.
2. Look further into the LightGBM documentation

Day 6

Ideas

- Our feeble attempts at adding polynomial and interactions have been largely unsuccessful, producing zero or negative gains in the overall AUC score. We are going to spend a large portion of our kaggle-allocated time today simply looking at the data. The aim of this task is to find anything that jumps out at us and hopefully make one or more interaction terms that yield significant gains.
- The highest correlations are between features `f8`, `f13` and `f19` (~0.2). Although this isn't much we will attempt to decompose these into one feature, if only to see what happens. Our hunch is that this is not right; the features are too disparate to be merged like this. Still, we will remain hopeful that at least `f8` (which is the least important of the three) may be completely replaced by this combination `f8wf13wf19`.
- There are multiple columns that seem to hover at the 110000 range. Of these `f4`, `f7`, `f13` and `f16` seem to like staying strictly in that level, while `f8`, `f17` and `f19` seem to like go up to the 300000 level as well. The trend is too prevalent to go ignored.
- We are going to create difference style interactions between a few of the above mentioned columns. There are a few interesting bits of knowledge that come to the fore: `f4` and `f7` seem to like staying within 1 or 2 of each other for ages, but then jump up to large differences.

```python
def transform(df, y=True):
    df_copy = df.copy()
    df_copy['f8-f19'] = df['f8'] - df['f19']
    df_copy['f8-f13'] = df['f8'] - df['f13']
    df_copy['f17-f4'] = df['f17'] - df['f4']
    df_copy['f4-f7'] = df['f4'] - df['f7']
    df_copy['f13-f19'] = df['f13'] - df['f19']
    df_copy['f8wf13wf19'] = PCA(n_components=1).fit_transform(df[['f8', 'f13',
'f19']])
    return df_copy.drop('Y', axis='columns') if y else df_copy
```

- We are going to select columns in a complex manner in which we combine the natural columns with the the synthetic columns we just made. Next, we classify them further into "veterans" and "ignored" depending on our confidence of their place. Then, loop through all the number of columns that we have, and at each step we:
    - step forward and select the column that along with the already selected columns gives the highest value of AUC-ROC
    - step backward and remove each of the columns within our chosen columns, introducing one of the other columns in its place to see if the combination gives

a better roc-auc score. If such a combination exists, set this as the new column set and repeat this step

```
veteran_cols = ['f14', 'f0', 'f13'] # features that we are confident about
ignored_cols = ['f5', 'f9', 'f11', 'f18', 'f19', 'f21'] # features that we know
couldn't possibly be right

def feature_selector(df):
    primary = (veteran_cols, 0)
    cols = np.setdiff1d(df.columns.tolist(), ignored_cols).tolist()
    for _ in range(len(cols) - len(veteran_cols)):
        best_feature_score = (0, 0)
        remaining_features = np.setdiff1d(cols, primary[0]).tolist()
        for current_feature in remaining_features:
            current_features = primary[0] + [current_feature]
            current_score = scorer(current_features)
            if current_score > best_feature_score[1]:
                best_feature_score = (current_feature, current_score)
        if best_feature_score[1] > primary[1]:
            primary = (primary[0] + [best_feature_score[0]],
best_feature_score[1])
            print(primary)
        else:
            break

    print('done')
    print(primary)
```

```
done.
['f14', 'f13', 'f15', 'f4', 'f8wf13wf19', 'f4-f7', 'f16', 'f17', 'f19', 'f1',
'f8-f19']
```

- Use LightGBM and our favoured parameters with the newly selected columns.

Day's best: 0.89395
Overall best: 0.89395

Reflections

What went well

- Got a miniscule in overall score (in the fourth decimal place). Our intuition is that this is hiding a higher increase on the private leaderboard because the local score increase was almost 0.05.
- Have a much more initimate understanding of the features now.

What we learnt

- PCA is almost unusable. It gives an increase in the AUC score but the low correlation between each of the columns (<= 0.2) means that, at least theoretically, we are losing data when we do that.
- Some of the difference between the columns are strongly correlated with each other (~0.9). The column sets are (f8, `f8-f13`, `f8-f19`) and (f17, `f17-f4`). It makes next to

no sense to us at this moment that the *differences* between these features are behaving in this manner. This lends credence to the possibility that the data is at least partially manually generated. We cannot see a way of taking advantage of this fact at the moment beyond the obvious of selecting only one of the features in these sets or decomposing them to generate a lower number of better features.

How we can improve

- At this point we are approaching the end of the competition placed around the top 5, and we have to believe that just about everyone has tried everything we have. That is, we need to try something not everyone would have done.
- Examining [this (https://www.kaggle.com/c/otto-group-product-classification-challenge/discussion/14335)](https://www.kaggle.com/c/otto-group-product-classification-challenge/discussion/14335) post of a past Kaggle winner reveals that after selecting good features, it is necessary to do something more involved to get the best possible AUC score. We will attempt true ensembling next.

Day 7

Ideas

- The first layer of the ensemble will use a combination of the raw features that we found performed best in addition to the more successful features that we synthesized the previous day. The second layer of the ensemble will take as input the soft output of the first layer models. The best features overall will be steadily added to the second layer input if they performance. The features used for the first layer of models were `f1`, `f4`, `f8`, `f13`, `f14`, `f15`, `f16`, `f17`, `f8-f19`, `f8-f13`, `f17-f4`, `f4-f7` and `f8wf13wf19`.
- Our ensemble will have a first layer of models that perform well or at least decently. It could be noted that the AUC score for the `KNeighborsClassifiers` peaked with 128 neighbors. The features for the second layer model were:
    - `RandomForestClassifier`
    - `ExtraTreesClassifier`
    - `AdaBoostClassifier`
    - `GradientBoostingClassifier`
    - `KNeighborsClassifier` with 64 neighbours
    - `KNeighborsClassifier` with 128 neighbours
    - `KNeighborsClassifier` with 256 neighbours
    - `LGBMClassifier`
    - `XGBClassifier`
    - And the same features as above added in order of importance (gain-wise) if they increase the AUC score.
- The first layer model parameters are to be found by `GridSearchCV`.

```
rf_params = {
    'class_weight': 'balanced',
    'criterion': 'gini',
    'n_estimators': 600
}

et_params = {
    'class_weight': 'balanced',
    'criterion': 'entropy',
    'n_estimators': 500
}

ada_params = {
    'n_estimators': 400,
    'learning_rate' : 0.9
}

gb_params = {
    'loss': 'exponential',
    'max_depth': 5,
    'n_estimators': 750,
}

lgb_params = {
    'boosting_type':                  'goss',
    'metric':                         'auc',
    'objective':                      'binary',
    'scale_pos_weight':               scale_pos_weight,
    'n_estimators':                   550,
    'learning_rate':                  0.09,
    'colsample_bytree':               0.1,
    'reg_alpha':                      0,
    'reg_lambda':                     0.89995,
}

xgb_params = {
    'objective':                       'binary:logistic',
    'booster':                         'gbtree',
    'tree_method':                     'exact',
    'eval_metric':                     'auc',
    'scale_pos_weight':                scale_pos_weight,
    'max_depth':                       12,
    'learning_rate':                   0.075,
    'n_estimators':                    350,
    'colsample_bylevel':               0.03,
    'colsample_bynode':                0.86
}
```

- And finally the second layer output would not be calibrated via a `scale_pos_weight` but instead using a dedicated calibrator `CalibratedClassifierCV`. The aim here is to counteract the confidence loss the second layer model would suffer when compared to the best first layer models that were exposed to the raw data.

```
train, x_cal, y, y_cal = train_test_split(train, y, test_size=0.2)

first_layer_models = [
    RandomForestClassifier(**rf_params),
    ExtraTreesClassifier(**et_params),
    AdaBoostClassifier(**ada_params),
    GradientBoostingClassifier(**gb_params),
    lgbm.LGBMClassifier(**lgb_params),
    xgb.XGBClassifier(**xgb_params)
]

[model.fit(train, y) for model in first_layer_models]

first_layer_train_preds = [model.predict_proba(train)[:, 1] for model in
first_layer_models]
first_layer_test_preds = [model.predict_proba(test)[:, 1] for model in
first_layer_models]
first_layer_cal = [model.predict_proba(x_cal)[:, 1] for model in
first_layer_models]

second_layer_train = np.append(np.array(first_layer_train_preds), train.T,
axis=0).T
second_layer_test = np.append(np.array(first_layer_test_preds), test.T,
axis=0).T
second_layer_cal = np.append(np.array(first_layer_cal), x_cal.T, axis=0).T

second_layer_model = xgb.XGBClassifier(**xgb2_params)
second_layer_model.fit(second_layer_train, y)

calibrator = CalibratedClassifierCV(second_layer_model, method='sigmoid',
cv='prefit')
calibrator.fit(second_layer_cal, y_cal)

last_predictions = calibrator.predict_proba(test)[:, 1]
```

Day's best: 0.89395
Overall best: 0.89395

Reflections

What went well

- We were able to find several pre-built ensembles to use in the first layer of models that gave very strong AUC scores of around 0.86 on average. The best of these, of course, was our previous best `LGBMClassifier` with the correct parameters with a score of ~0.89.
- Very curiously, we obtained exactly the same AUC score as our best model doing the whole setup.

What we learnt

- A key concept here during training is to use a `StratifiedKFold` cross validator with a constant `random_state` set so that it never occurs that the first layer models have already "seen" the test data of the second layer model. We had an AUC score of 0.99 the first time testing the setup which was actually an unfortunate 0.82 on the true test set on Kaggle.

- It turns out that the ensemble did best while using all the first layer features as an additional input to the second layer. This was unexpected too. Fortunately, looking at the `features_importances_` attribute of the second layer model revealed that it was relying on the first layer outputs too.
- The fact that the ensemble did only as well as the best first layer model and required the same inputs to do so throws us off here. We were expecting our attempt to probably fail and hopefully do extremely well. The fact that it performed the same means that the purpose of ensembling is lost. It is possible that the second layer model "detected" the output of the best performer and relied on its output more than it did the other first layer models.
- We need much more experience with machine learning to attempt proper stacking and ensembling.