

Allan Frederick

EE460J: Data Science Lab

26 October 2020

Kaggle Report

Exploratory Data Analysis

Before doing anything else, the first step I took was to just look at the data, including analyzing a graphical representation of the features.

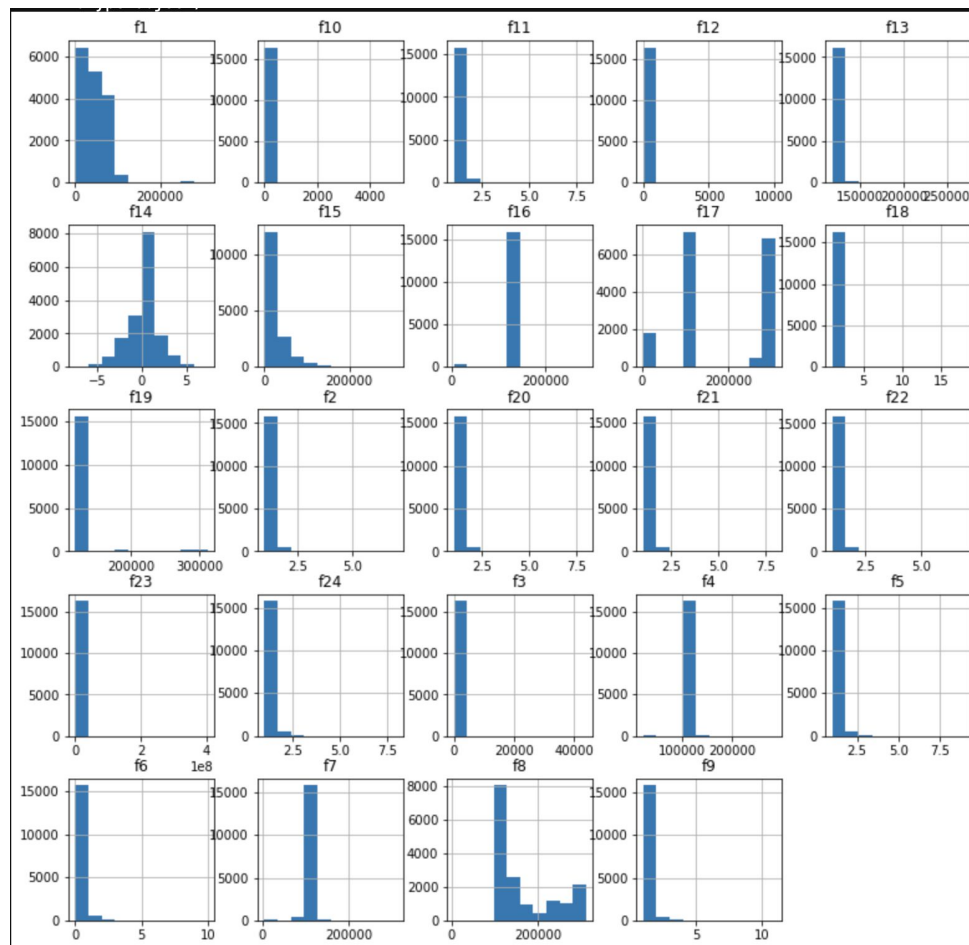


Figure 1. Histogram of feature data.

These are the exploratory data analysis conclusions I came up with:

- No missing values in the dataset
- Many features are categorical
- Categorical features appear to be:
f2,f3,f4,f5,f6,f7,f9,f11,f12,f13,f16,f17,f18,f19,f20,f21,f22,f23,f24

- Non categorical features appear to be: f1,f8,f14,f15

Knowing which features are categorical and which are not was worth noting because it could help with training certain models. A feature was determined to be categorical by estimation, specifically by looking at how the data in the feature was distributed from the histogram.

Features that clearly looked like they were placed in “bins” rather than spread across the axis, I considered to be categorical.

What I Tried

- XGBoost Classifier with raw data
- XGBoost Classifier with raw data trained with only features considered the “most important”
- XGBoost Classifier with dummified data
- Random Forests Classifier
- Catboost with raw data
- Catboost using categorical features
- Various Ensemble stackings

As mentioned in class, using XGboost alone without much preprocessing should yield a decent result, so it was the first model I trained. I specifically did not train any neural networks or logistic regression models since I heard from multiple sources that those returned poor results, and I wanted to use my time as efficiently as possible. After tuning XGBoost alone I got pretty decent results, so I decided that decision trees were the way to go. But unfortunately, there were certain obstacles along the way which limited my ability to ultimately train the final model that I wanted.

What Worked Well - (Kaggle public score > 0.90)

XGBoost Classifier with raw data:

The first model I trained was an XGBoost. To tune this model, I used a series of grid searches to find the optimal hyperparameters. I found great online resources that guided me through tuning the XGB model. Finding all hyperparameters at once by calling a single instance of gridsearch would take an incredible amount of time that I did not have, so I decided it would be more efficient to grid search a few hyperparameters at a time, starting first with the highest impact parameters. I also set the learning rate significantly higher and the number of estimators significantly lower than what I would train my final model with for time saving purposes. I

would decrease my learning rate and increase estimators last, after the rest of the hyperparameters have been optimized. I didn't want to overdo it however, as increasing the estimators too much could've led to overfitting. This was the general strategy for hyperparameter tuning I learned from the article.

I used this same strategy to tune all my other models. It took about 5 grid search runs to optimize the XGB model, resulting in a roc_auc score of 0.8831224337343377. To verify this, I ran cross validation on the model, separate from what was used in grid search. I did this because although gridsearch uses stratifiedKfold for cross validation, the default parameter of [shuffle = False] is passed, so the samples are not actually shuffled. Grid search results are most likely reliable, but having the samples shuffled when doing cross validation is a way to ensure if the model is actually good or not, as I learned in class. The corresponding kaggle public score using this model was 0.90239.

```
XGB parameter tuning (without dummified X_train)

: # First tune the highest impact parameters
: # Typical initial estimators for everything else

param_test1 = {
    'max_depth': range(3, 11, 2),
    'min_child_weight': range(1, 7, 2)
}

gsearch1 = GridSearchCV(
    estimator = XGBClassifier(
        learning_rate = 0.1,
        n_estimators = 140,
        gamma = 0,
        subsample = 0.8,
        colsample_bytree = 0.8,
        objective = 'binary:logistic',
        scale_pos_weight = 1,
        seed=42),
    param_grid = param_test1,
    scoring='roc_auc',
    n_jobs=-1,
    cv=5)

gsearch1.fit(X_train, y_train.values.ravel())

: gsearch1.best_params_, gsearch1.best_score_

: ({'max_depth': 9, 'min_child_weight': 1}, 0.8730321403335572)
```

Figure 2. Initial grid search of XGB

```

# Search for 1 above and below opt parameters 1 because previous values were searched in intervals of 2

param_test2 = {
    'max_depth':[8,9,10],
    'min_child_weight':[0,1,2]
}

gsearch2 = GridSearchCV(
    estimator = XGBClassifier(
        learning_rate = 0.1,
        n_estimators = 140,
        gamma = 0,
        subsample = 0.8,
        colsample_bytree = 0.8,
        objective = 'binary:logistic',
        scale_pos_weight = 1,
        seed=42),
    param_grid = param_test2,
    scoring='roc_auc',
    n_jobs=-1,
    cv=5)

gsearch2.fit(X_train,y_train.values.ravel())

gsearch2.best_params_, gsearch2.best_score_

({'max_depth': 10, 'min_child_weight': 2}, 0.8741829477411631)

```

Figure 3. Second grid search of XGB

```

cross_val_score(xgb_model, X_train, y_train.values.ravel(),
                cv=sklearn.model_selection.StratifiedKFold(n_splits=5, shuffle=True),
                scoring="roc_auc")

array([0.87680937, 0.89609568, 0.8870646 , 0.87903874, 0.87899246])

```

Figure 4. Cross validation of final XGB model for verification

Catboost:

Following the same hyperparameter tuning procedure as with XGBoost, Catboost turned out to be the best performing model. To make sure I was on the right track in terms of choosing the optimal parameters, I ran grid searches trying multiple cv's in addition to making sure that samples were shuffled.

```

scoring='roc_auc',
n_jobs=-1,
# use shuffle to verify validity of hyperparameters
cv= sklearn.model_selection.StratifiedKFold(n_splits=5, shuffle=True)
)

cat_gsearch1.fit(X_train,y_train.values.ravel())

j: cat_gsearch1.best_params_, cat_gsearch1.best_score_

cv = 5: ({'depth': 7, 'min_data_in_leaf': 1}, 0.8560175950079281)

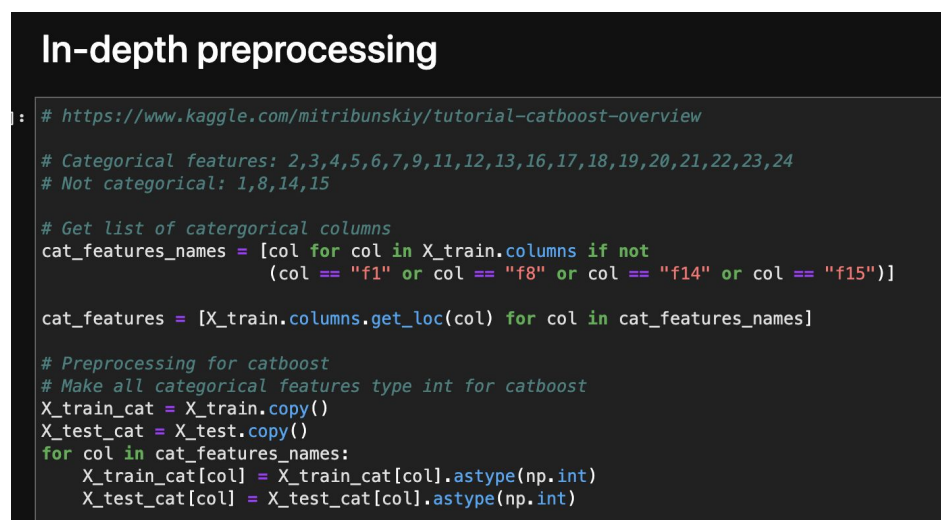
cv = 10: ({'depth': 7, 'min_data_in_leaf': 1}, 0.8633634575626921)

Shuffle = True -> provides a better metric for seeing how good model is

```

Figure 5. Initial grid search of Catboost

The best score using only the raw training data was below 0.87 for multiple grid search runs. Unlike XGBoost, however, that only supports numerical features, Catboost also supports categorical features. This can be specified by passing in the parameter “cat_features”, which is a list of all the indices of the categorical features in the data. According to the Catboost documentation, one-hot-encoding is performed by default on categorical features. Utilizing this would further improve the model score. After following a tutorial and having a list which features appear to be categorical, all I had to do to utilize cat_features was to just convert all categorical data into integer types and get their location in the training data.



```
In-depth preprocessing

: # https://www.kaggle.com/mitribunskiy/tutorial-catboost-overview

# Categorical features: 2,3,4,5,6,7,9,11,12,13,16,17,18,19,20,21,22,23,24
# Not categorical: 1,8,14,15

# Get list of categorical columns
cat_features_names = [col for col in X_train.columns if not
                      (col == "f1" or col == "f8" or col == "f14" or col == "f15")]

cat_features = [X_train.columns.get_loc(col) for col in cat_features_names]

# Preprocessing for catboost
# Make all categorical features type int for catboost
X_train_cat = X_train.copy()
X_test_cat = X_test.copy()
for col in cat_features_names:
    X_train_cat[col] = X_train_cat[col].astype(np.int)
    X_test_cat[col] = X_test_cat[col].astype(np.int)
```

Figure 6. Preprocessing for Catboost

Passing in cat_features to specify my categorical data into the model significantly improved the score. I was now reaching the 0.90’s when passing in cat_features, compared to the 0.86 range when training only with numerical data. The kaggle public score for catboost was 0.91667.

```

# Creating a catboost model using optimal parameters

CATparams = {
    'learning_rate' : 0.01,
    'depth' : 6,
    'iterations' : 800,
    'loss_function' : 'Logloss',
    'border_count' : 32,
    'eval_metric' : "AUC",
    'min_data_in_leaf' : 1,
    'cat_features' : cat_features,
    'random_seed' : 42,
    'silent' : True
}

cat_model = CatBoostClassifier(**CATparams)
cat_model.fit(X_train_cat, y_train.values.ravel())

<catboost.core.CatBoostClassifier at 0x12a70e130>

cross_val_score(cat_model, X_train_cat, y_train.values.ravel(),
                 cv=sklearn.model_selection.StratifiedKFold(n_splits=5, shuffle=True),
                 scoring="roc_auc")

array([0.89078649, 0.90327178, 0.89123319, 0.90565488, 0.89036633])

```

Figure 7. Final Catboost parameters and cv score

Stacking- XGB, Random Forests, XGB as final estimator

The only other model that returned a public kaggle score greater than .90 was an ensemble stacking model combining XGB and random forests, with XGB as the final estimator. All the models used were the previous models I optimized using gridsearch. However, this is when I started running to problems, as I was unable to stack certain models. This is further elaborated on in the **Challenges** section.

What Did Not Work Well (Kaggle public score < 0.90)

XGBoost with only “important” features

I tried to create a model that considered only the most important features. Ideally if it worked, I would’ve combined it with a stacking to see if that would improve the score. First I just wanted to get a quick visual of which features were the most important. Luckily XGB has a built in function to plot feature importance based on the f-score of each feature. This was reliable because the initial XGB performed quite decently. Specifically, I was able to obtain the “optimal features” using a feature engineering function from SKlearn called RFEC. This stands for recursive feature elimination and cross-validated selection, and is used for feature ranking.

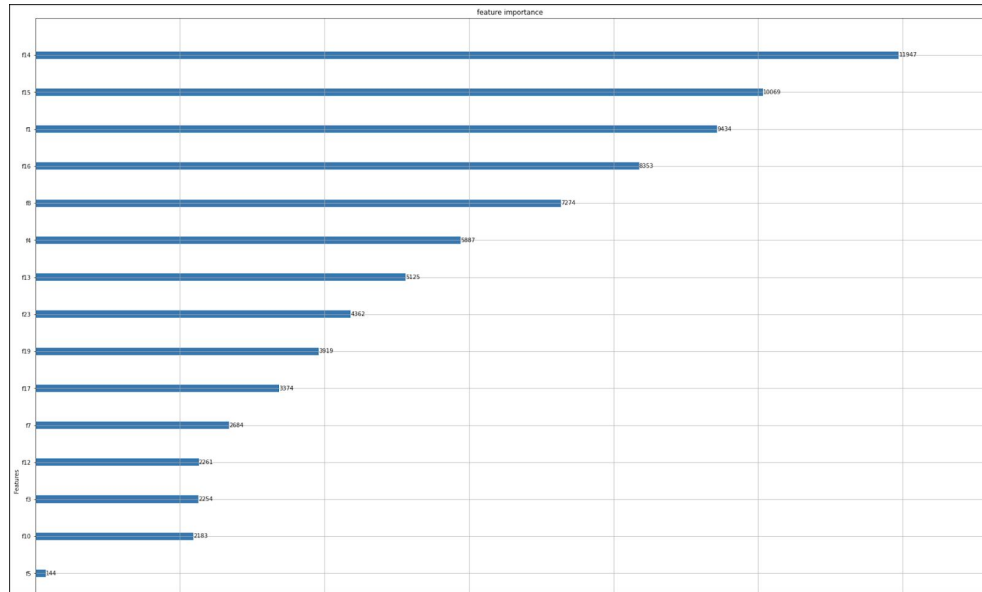


Figure 8. Feature importance plot from XGB

```

: # Find optimal features
# https://www.kaggle.com/kanncaal/feature-selection-and-data-visualization#Feature-Selection-and-Random-For

# The "accuracy" scoring is proportional to the number of correct classifications
rfecv = RFECV(estimator=xgb_model, step=1, cv=5, scoring='accuracy', n_jobs=-1) #5-fold cross-validation
rfecv = rfecv.fit(X_train, y_train.values.ravel())

print('Optimal number of features :', rfecv.n_features_)
print('Best features :', X_train.columns[rfecv.support_])
print("Accuracy: ", rfecv.score(X_train, y_train))
os.system("say 'bingo'")

X_opt = rfecv.transform(X_train)
X_opt_df = pd.DataFrame(X_opt)

xgb_OPT_model = XGBClassifier(**XGBparams)
xgb_OPT_model.fit(X_opt_df, y_train.values.ravel())
cross_val_score(xgb_OPT_model, X_train, y_train.values.ravel(),
                 cv=sklearn.model_selection.StratifiedKFold(n_splits=5, shuffle=True),
                 scoring="accuracy")

Optimal number of features : 9
Best features : Index(['f1', 'f4', 'f7', 'f8', 'f14', 'f15', 'f16', 'f17', 'f19'], dtype='object')
Accuracy: 0.9775376915094915
: array([0.89011474, 0.87962849, 0.87647009, 0.87715424, 0.88196448])

```

Figure 9. RFECV implementation

This model gave pretty decent scores for cross validation, but when implemented as a model in ensemble stacking, the model performed worse than not including it. Therefore, I just assumed that it would similarly return a worse score on kaggle and did not submit it for a public kaggle score.

Random Forests Classifier

Another decision tree model that I used was Random Forests. The general strategy I used to tune hyperparameters also applies to tuning this model. The cross validation scores were decent, but worse than XGB and Cat. The public kaggle score for this model alone was 0.88111. However, the model was used in an ensemble stacking so it may have positively contributed to that score.

```
# Creating a model using optimal parameters

RfcParams = {
    'n_estimators': 1500,
    'min_samples_split': 5,
    'min_samples_leaf': 2,
    'max_features': 'auto',
    'max_depth': 30
}

rfc_model = RandomForestClassifier(**RfcParams, random_state = 42)
rfc_model.fit(X_train, y_train.values.ravel())

RandomForestClassifier(max_depth=30, min_samples_leaf=2, min_samples_split=5,
                       n_estimators=1500, random_state=42)

cross_val_score(rfc_model, X_train, y_train.values.ravel(),
                 cv=sklearn.model_selection.StratifiedKFold(n_splits=5, shuffle=True),
                 scoring="roc_auc")

array([0.86394387, 0.85623412, 0.85799362, 0.87932669, 0.85759534])
```

Figure 10. Random Forest Classifier CV scores

XGB with dummified train data

Since XGB was one of the best performing models, I wanted to see if I could use my time to improve it even more. Since XGB only deals with numerical data, and the training data we were given appeared to also be categorical, I thought it would be beneficial to dummify the categorical data using `get_dummies` from pandas. Doing this created a significant number of extra columns. There were 2916 columns after encoding the categorical data. When passing `get_dummies`, there is an option to drop the first level of categorical variables, which would decrease the number of extra columns. In some instances, this should be used, but I read on a Stack Exchange forum that keeping all the dummies for a decision tree model is actually beneficial (reference is in the jupyter file).

	f1	f8	f14	f15	f2_1	f2_2	f2_3	f2_4	f2_5	f2_6	...	f23_17681416	f23_42640756	f23_404288627	f24_1	f24_2	1
0	25884	125738	-2.266430	1945	1	0	0	0	0	0	...	0	0	0	1	0	
1	34346	130913	-0.305612	15385	1	0	0	0	0	0	...	0	0	0	1	0	
2	34923	124402	2.015561	7547	1	0	0	0	0	0	...	0	0	0	1	0	
3	80926	301218	-3.172501	4933	1	0	0	0	0	0	...	0	0	0	1	0	
4	4674	302830	0.573767	13836	1	0	0	0	0	0	...	0	0	0	1	0	
...	
16378	33328	117906	1.000000	4533	1	0	0	0	0	0	...	0	0	0	1	0	
16379	19944	132097	1.000000	4663	1	0	0	0	0	0	...	0	0	0	1	0	
16380	28359	174445	1.806069	7822	1	0	0	0	0	0	...	0	0	0	1	0	
16381	7542	117906	-1.245079	6257	1	0	0	0	0	0	...	0	0	0	1	0	
16382	13877	117879	3.626300	13262	1	0	0	0	0	0	...	0	0	0	1	0	

16383 rows x 2916 columns

Figure 11. Dummified dataframe

XGB with dummy X_train

Used OPT paramters found with X_train normal to save time

```
# Using dummy X_train (preprocessed)

param_test_xgb = {}

gsearchXGB = GridSearchCV(
    estimator = XGBClassifier(
        max_depth = 10,
        min_child_weight = 2,
        learning_rate = 0.01,
        n_estimators = 1000,
        gamma = 0.1,
        subsample = 0.8,
        colsample_bytree = 0.5,
        objective = 'binary:logistic',
        scale_pos_weight = 1,
        seed=42),
    param_grid = param_test_xgb,
    scoring='roc_auc',
    n_jobs=-1,
    cv=5)

gsearchXGB.fit(df, y_train.values.ravel())
gsearchXGB.best_params_, gsearchXGB.best_score_

({}, 0.8671792256439248)
```

Figure 12. XGB with dummified data CV score

As much as I thought using `get_dummies` on the training data would improve the XGB model, it actually performed much, much worse. Either that or I incorrectly applied `get_dummies` on the data. The public kaggle score returned a 0.81894, which was disappointing, especially because it took quite a long time to fit and cross validate this model.

Challenges (What did not work at all)

Catboost Stacking

Catboost was my best performing model. That's why it made sense for me to incorporate that into an ensemble stacking. This was also the biggest obstacle. I was unable to do any kind of stacking involving Catboost. Catboost performed the best on the training data that was preprocessed in such a way that converted categorical data to integer types. The XGB model performed the best on the raw training data, without any pre-processing. I was unable to combine these models.

Initially I tried incorporating Catboost as both a stacked estimator and final estimator. The training data I used to train the final estimator was using the same pre-processed data that was used to train the standalone Catboost model. This error was returned:

```
1794         group_weight, subgroup_id, pairs_weight, baseline,
~/anaconda3/lib/python3.8/site-packages/catboost/core.py in _prepare_train_params(self, X, y, cat_features, text_features, embedding_features, pairs, sample_weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, use_best_model, eval_set, verbose, logging_level, plot, column_description, verbose_eval, metric_period, silent, early_stopping_rounds, save_snapshot, snapshot_file, snapshot_interval, init_model)
1680         embedding_features = _process_feature_indices(embedding_features, X, params, 'embedding_features')
1681
-> 1682         train_pool = _build_train_pool(X, y, cat_features, text_features, embedding_features, pairs,
1683                                     sample_weight, group_id, group_weight, subgroup_id, pairs_weight,
1684                                     baseline, column_description)
~/anaconda3/lib/python3.8/site-packages/catboost/core.py in _build_train_pool(X, y, cat_features, text_features, embedding_features, pairs, sample_weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, column_description)
982         if y is None:
983             raise CatBoostError("y has not initialized in fit(): X is not catboost.Pool object, y must be not None in fit().")
-> 984         train_pool = Pool(X, y, cat_features=cat_features, text_features=text_features, embedding_features=embedding_features, pairs=pairs, weight=sample_weight, group_id=group_id,
985                         group_weight=group_weight, subgroup_id=subgroup_id, pairs_weight=pairs_weight, baseline=baseline)
986         return train_pool
~/anaconda3/lib/python3.8/site-packages/catboost/core.py in __init__(self, data, label, cat_features, text_features, embedding_features, column_description, pairs, delimiter, has_header, ignore_csv_quoting, weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, feature_names, thread_count)
416         elif isinstance(data, np.ndarray):
417             if (data.dtype.kind == 'f') and (cat_features is not None) and (len(cat_features) > 0):
-> 418                 raise CatBoostError(
419                     "'data' is numpy array of floating point numerical type, it means no categorical features,"
420                     " but 'cat_features' parameter specifies nonzero number of categorical features"
CatBoostError: 'data' is numpy array of floating point numerical type, it means no categorical features, but 'cat_features' parameter specifies nonzero number of categorical features
```

Figure 13. CatboostError #1

After bringing this issue to a TA, we concluded that it is the predicted probabilities that are passed to the final model for training of the final estimator. This is specified by the parameter `stack_method`. By default, it is set to "auto" which passes 'predict_proba'. This is not an integer, but Catboost data requires integers to be passed for categorical data. To fix this, I set the `stack_method` to 'predict', meaning that the hard predictions would be passed onto the model, ensuring that the data is of integer type. Then I got another error:

```

one in fit().")
--> 984     train_pool = Pool(X, y, cat_features=cat_features, text_features=text_features, embedding_features=embedding_features, pairs=pairs, weight=sample_weight, group_id=group_id,
985                        group_weight=group_weight, subgroup_id=subgroup_id, pairs_weight=pairs_weight, baseline=baseline)
986     return train_pool

~/anaconda3/lib/python3.8/site-packages/catboost/core.py in __init__(self, data, label, cat_features, text_features, embedding_features, column_description, pairs, delimiter, has_header, ignore_csv_quoting, weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, feature_names, thread_count)
453         )
454     --> 455         self._init(data, label, cat_features, text_features, embedding_features, pairs, weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, feature_names, thread_count)
456         super(Pool, self)._init__()
457

~/anaconda3/lib/python3.8/site-packages/catboost/core.py in _init(self, data, label, cat_features, text_features, embedding_features, pairs, weight, group_id, group_weight, subgroup_id, pairs_weight, baseline, feature_names, thread_count)
924     cat_features = _get_features_indices(cat_features, feature_names)
925     self._check_string_feature_type(cat_features, 'cat_features')
--> 926     self._check_string_feature_value(cat_features, features_count, 'cat_features')
927     if text_features is not None:
928         text_features = _get_features_indices(text_features, feature_names)

~/anaconda3/lib/python3.8/site-packages/catboost/core.py in _check_string_feature_value(self, features, features_count, features_name)
495         raise CatBoostError("Invalid {}[{}] = {} value type={}: must be int().".format(features_name, index, feature, type(feature)))
496     if feature >= features_count:
--> 497         raise CatBoostError("Invalid {}[{}] = {} value: index must be < {}".format(features_name, index, feature, features_count))
498
499     def _check_pairs_type(self, pairs):

CatBoostError: Invalid cat_features[0] = 1 value: index must be < 1.

```

Figure 14. Catboost error #2

It seems that the predictions from previous trained models were being passed onto the final estimator, which was being received as more categorical features. But the final cat model being trained only knew of the categorical features created in `cat_features`, used in the training of the initial cat model, but this did not get passed to the final estimator. I somehow needed a way to insert new categorical data after the stacking models were trained, but before the final estimator was trained. To circumvent this issue, I then tried to change another parameter of ensemble stacking called “Passthrough”, setting it to true (false is the default value). When true, the final model would also be trained on the initial data used to train the stacking estimators. That way, the final estimator would be able to access the `cat_features`. This resulted in the initial catboost error I got (figure 13). I had to change a parameter to fix the initial error, but doing so created another error. To fix that error I had to change another parameter, but doing that caused the initial error to occur. It seemed as if I was stuck in a loop. Moreover, these errors occurred not only when Catboost was set as the final estimator, but also when it was just a stacked model. In other words, I was unable to incorporate Catboost in any way for stacked models. The Ensemble Stacking module is a bit of a black box, so the TA recommended using a pipeline, but this did not guarantee to fix the issue. I decided that my time would be better spent trying to improve something that was guaranteed to work; that is when I decided to apply `get_dummies` on the data for XGB, but it turned out to make the model worse.

Reflections

Ultimately, I am surprised that my best model was just Catboost. But it does make sense since Catboost was the only model that incorporated categorical features into its training. I am most surprised by XGB actually performing worse with the dummified data, which makes me think that I didn't use it properly. I think stacking XGB with Catboost (my two best models) would've performed the best, so I am disappointed that I couldn't find a way to do any kind of stacking with Catboost. I did end up selecting the right submissions for my private LB scoring, as these corresponded to the highest public LB scores. From this competition, I learned an effective strategy for hyperparameter tuning that can be applied to any model. I also learned ways to verify how good a model is in terms of cross validation, and specifically, the importance of trying several different cv numbers as well as shuffling samples.

```
# FINAL SCORES #  
  
### XGB (raw train) -> public: 0.90239, private: 0.89914  
  
### XGB (dummy train) -> public: 0.81894, private: 0.80905  
  
### RandomForest -> public: 0.88111, private: 0.88475  
  
### Catboost -> public: 0.91667, private: 0.90152  
  
### Stacking-xgb w/ rfc (raw train) -> public: 0.90360, private: 0.89688  
  
### Stacking-xgb w/ xgb -> public: 0.87941, private: 0.88489
```

Figure 15. Final scores returned from kaggle

Kaggle Public LB

36	Allan Frederick		0.91667
----	-----------------	---	---------

Kaggle Private LB

52	▼ 16 Allan Frederick		0.90152
----	----------------------	---	---------