Homework Assignment 7

Sara Clokey

Table of contents

The business problem at hand in this competition is predicting the number of times a policyholder at CloverShield Insurance is likely to call the call center. This prediction model will be used to lower call center costs and improve allocation of resources.

To begin approaching this problem, I first imported all necessary packages, including 'pandas,' 'sklearn,' and 'lightgbm'. I then read both the training and testing CSV files into the code.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from lightgbm import LGBMRegressor
from sklearn.metrics import mean_squared_error
import lightgbm as lgb

train_file_path = 'data/train_data.csv'
df_train = pd.read_csv(train_file_path)

test_file_path = 'data/test_data.csv'
df_test = pd.read_csv(test_file_path)
```

After introducing the training data set, I began cleaning and analyzing the data. First, using the variable definitions found in the competition, I searched for missing or invalid values. In the 'newest_veh_age' column, '-20' represents a non-auto or missing value, so I searched for how many values of this nature existed. For the 'telematics_ind' variable, '0' represents a missing value and '-2' represents a non-auto value, so I separate these distinctions and search for the number of instances of each circumstance. For 'pol_edeliv_ind' variable, '-2' represents a missing value, so I search for the number of rows where this occurs.

```
veh_missing_count = (df_train['newest_veh_age'] == -20).sum()
print(f"Number of missing values in 'newest_veh_age': {veh_missing_count}")

auto_missing_count = (df_train['telematics_ind'] == 0).sum()
```

```
non_auto_count = (df_train['telematics_ind'] == -2).sum()
print(f"Number of auto missing values in 'telematics_ind' (0): "
      f"{auto_missing_count}")
print(f"Number of non-auto values in 'telematics_ind' (-2): "
      f"{non_auto_count}")

pol_missing_count = (df_train['pol_edeliv_ind'] == -2).sum()
print(f"Number of missing values in 'pol_edeliv_ind' (-2): "
      f"{pol_missing_count}")
```

```
Number of missing values in 'newest_veh_age': 58015
Number of auto missing values in 'telematics_ind' (0): 7317
Number of non-auto values in 'telematics_ind' (-2): 58015
Number of missing values in 'pol_edeliv_ind' (-2): 838
```

Based on the results of those missing count searches, I decide to remove the 'newest_veh_age' variable because there are over 58,000 missing values. I then drop individual rows where 'telematics_ind' has a '0' or where 'pol_edeliv_ind' has a '-2'. Removing these rows will allow for more efficient and accurate data analysis.

```
# Data cleaning
df_train = df_train.drop('newest_veh_age', axis=1)
df_train = df_train.drop(df_train[df_train['telematics_ind'] == 0].index)
df_train = df_train.drop(df_train[df_train['pol_edeliv_ind'] == -2].index)
```

After the data is cleaned, I move to feature engineering based on the variables included in the data set. First, I create a variable called 'annualized_premium_per_policy,' which divides the annualized premium amount ('ann_prm_amt') by the number of policies in a household ('household_policy_count') to determine an annualized premium per household policy. This value is added to the training data frame and is rounded to two decimals. I also create a variable called 'multiple_policies_indicator,' which is a binary variable that denotes whether a household has more than one policy.

```
df_train['annualized_premium_per_policy'] = (
    df_train['ann_prm_amt'] / df_train['household_policy_counts']
).round(2)

df_train['multiple_policies_indicator'] = (
    (df_train['household_policy_counts'] > 1).astype(int)
)
```

Next, I use One-Hot Encoding from scikit-learn to prepare all of the categorical variables in the data set for modeling, which transforms them into binary columns for each possible item in the variable. I first print the data types for each variable in the training data frame, then use One-Hot Encoding for all variables with 'object' data type.

```
print(df_train.dtypes)
```

```
id                                 int64
12m_call_history                   int64
acq_method                        object
ann_prm_amt                      float64
bi_limit_group                    object
channel                           object
digital_contact_ind                int64
geo_group                         object
has_prior_carrier                  int64
home_lot_sq_footage                int64
household_group                   object
household_policy_counts            int64
pay_type_code                     object
pol_edeliv_ind                     int64
prdct_sbtyp_grp                   object
product_sbtyp                     object
telematics_ind                     int64
tenure_at_snapshot                 int64
trm_len_mo                         int64
call_counts                        int64
annualized_premium_per_policy    float64
multiple_policies_indicator        int64
dtype: object
```

```
df_train_encoded = pd.get_dummies(
    df_train,
    columns=[
        'acq_method',
        'bi_limit_group',
        'channel',
        'geo_group',
        'household_group',
        'pay_type_code',
        'prdct_sbtyp_grp',
        'product_sbtyp'
    ],
    drop_first=True
)
```

I encode the categorical variables in the testing data frame in the same way to provide continuity and efficiency when predicting on this data later in the code.

```
df_test_encoded = pd.get_dummies(
    df_test,
```

```
    columns=[
        'acq_method',
        'bi_limit_group',
        'channel',
        'geo_group',
        'household_group',
        'pay_type_code',
        'prdct_sbtyp_grp',
        'product_sbtyp'
    ],
    drop_first=True
)
```

After preparing the variables, I prepare the training data set for modeling by creating 'X_train' and 'y_train' labels. 'X_train' includes every explanatory variable, and 'y_train' includes the target variable, 'call_counts'.

```
X_train = df_train_encoded.drop(columns=['call_counts'])
y_train = df_train_encoded['call_counts']
```

To begin training the model, I use the LGMRegressor model from the lightgbm package. I chose this model because it is known for speed and scalability; given the size of this data set, these features are important. Additionally, LightGBM builds on a foundation created by boosted trees, which emphasizes efficacy based on error correction, weighted learning, and flexibility.

I also create a parameter grid with multiple options for the number of leaves, the learning rate (speed of the model), the number of boosting trees, maximum depth of each tree, the minimum samples required to create a branch node, the percentage of samples used to fit an individual tree, the fraction of features considered when building a tree, and the fraction of data used for each iteration. I use GridSearchCV to find the best parameters in the grid based on the 'X_train' and 'y_train' data. These favorable parameters are printed as outputs.

```
model = LGBMRegressor(random_state=3486)

param_grid = {
    'num_leaves': [31, 50],
    'learning_rate': [0.01, 0.1],
    'n_estimators': [100, 200],
    'boosting_type': ['gbdt'],
    'max_depth': [-1, 10],
    'min_child_samples': [10, 20],
    'subsample': [0.8, 1.0],
    'feature_fraction': [0.5, 0.8],
    'bagging_fraction': [0.5, 0.8],
}

grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
```

```
                           scoring='neg_mean_squared_error', cv=5, n_jobs=-1)

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
best_score = (-grid_search.best_score_) ** 0.5
print(f"Best Parameters: {best_params}")
print(f"Best Cross-Validated RMSE: {best_score:.2f}")
```

```
[LightGBM] [Warning] feature_fraction is set=0.8, colsample_bytree=1.0 will be ignored. Current v
[LightGBM] [Warning] bagging_fraction is set=0.5, subsample=0.8 will be ignored. Current value: b
[LightGBM] [Warning] feature_fraction is set=0.8, colsample_bytree=1.0 will be ignored. Current v
[LightGBM] [Warning] bagging_fraction is set=0.5, subsample=0.8 will be ignored. Current value: b
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.004637 se
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1408
[LightGBM] [Info] Number of data points in the train set: 71913, number of used features: 36
[LightGBM] [Info] Start training from score 25.859400
Best Parameters: {'bagging_fraction': 0.5, 'boosting_type': 'gbdt', 'feature_fraction': 0.8, 'lea
Best Cross-Validated RMSE: 36.00
```

Using the best parameters found in the previous code block, I build a LightGBM regression model that is fit on the 'X_train' and 'y_train' data.

I then prepare the testing data set with labels 'X_test' for the explanatory variables. There is no 'call_count' variable in the training set, so the 'X_test' label includes all of the variables in the encoded version of the testing data. The 'X_test' and 'X_train' sets are compared to ensure they have the same features.

The 'best_model' created before is then used to predict the values of y, or 'call_counts,' in the testing data using the 'X_test' data and based on the LightGBM Model built by the training data.

These 'call_counts' predictions are saved to a data frame and, subsequently, to a CSV file to be uploaded to the competition site.

```
best_model = LGBMRegressor(**best_params)
best_model.fit(X_train, y_train)

X_test = df_test_encoded

X_test = X_test.reindex(columns=X_train.columns, fill_value=0)

y_test_pred = best_model.predict(X_test)
```

```python
predictions_df = pd.DataFrame({
    'id': range(1, len(y_test_pred) + 1),
    'Predict': y_test_pred
})

output_file_path = 'predictions.csv'
predictions_df.to_csv(output_file_path, index=False)

print(f"Predictions saved to {output_file_path}")
```

```
[LightGBM] [Warning] feature_fraction is set=0.8, colsample_bytree=1.0 will be ignored. Current v
[LightGBM] [Warning] bagging_fraction is set=0.5, subsample=0.8 will be ignored. Current value: b
[LightGBM] [Warning] feature_fraction is set=0.8, colsample_bytree=1.0 will be ignored. Current v
[LightGBM] [Warning] bagging_fraction is set=0.5, subsample=0.8 will be ignored. Current value: b
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.004933 se
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 1408
[LightGBM] [Info] Number of data points in the train set: 71913, number of used features: 36
[LightGBM] [Info] Start training from score 25.859400
[LightGBM] [Warning] feature_fraction is set=0.8, colsample_bytree=1.0 will be ignored. Current v
[LightGBM] [Warning] bagging_fraction is set=0.5, subsample=0.8 will be ignored. Current value: b
Predictions saved to predictions.csv
```

When submitted to the competition page and evaluated using the Gini Index, this model receives a score of 0.25502, which is 0.0029 better than the Travelers Benchmark Model score of 0.25212. This improvement can be explained by the feature engineering and parameter tuning in my LightGBM model that is not present in the benchmark model.