FinalModel

November 16, 2020

Import Required Modules Below I collect the tools that I will use to build the model. After initial exploratory modeling I found that `XGBClassifier` provided the best performance, measured in terms of model accuracy.

```
[ ]: #Basics
import pandas as pd
import numpy as np

#Plotting
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_context('notebook')

#Train Test Split
from sklearn.model_selection import train_test_split

# Imputer
from sklearn.impute import SimpleImputer

# Preprocessing
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import (
    OneHotEncoder,
    StandardScaler,
    FunctionTransformer)

# Classifiers
from xgboost import XGBClassifier
from sklearn.multiclass import OneVsRestClassifier

#Pipeline
from sklearn.pipeline import Pipeline

#Grid Search
from sklearn.model_selection import GridSearchCV

# Model evaluation
from sklearn.metrics import plot_confusion_matrix
```

Set Random State There are a few steps below where random processes require a seed. For reproducibility, I set a default random state below.

```
[ ]: random_state = 42
```

Select Columns to Drop from the Model Provide a list of variables that should be dropped from the model. I have not observed any improvement in model performance from dropping data, measured in terms of accuracy. Training time is obviously improved by dropping columns but there seems to be a small price to pay in terms of accuracy for reducing the number of available features.

```
[ ]: drop_cols = []
```

Import Data Because of the submission format requirements for the competition, it is vital that I retain the index column through out modeling so that I are able to produce predictions that can be validated using the competition's validation data.

```
[ ]: features = pd.read_csv(
         filepath_or_buffer='../data/training_features.csv',
         index_col='id'
     )
     targets = pd.read_csv(
         filepath_or_buffer='../data/training_labels.csv',
         ndex_col='id'
                          )
     df = features.join(targets, how='left')
     X = df.drop('status_group', axis=1)
     y = df['status_group']
```

Make Test Train Split For the purposes of model tuning I hold 10% of the data out for local testing.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(
         X,
         y,
         test_size=0.1,
         random_state=random_state
     )
```

Data Validation I experimented with both manual and automated feature selection, hoIver neither approach improved model performance. Initially, I had issues with mixed data types in both the `public_meeting` and `permit` columns. The function below converts all categorical variables to strings to eliminate thoes errors.

```
[ ]: def convert_categorical_to_string(data):
         return pd.DataFrame(data).astype(str)
```

```
CategoricalTypeConverter = FunctionTransformer(
    convert_categorical_to_string
)
```

Classify Variables I will need to pre-process the data in preparation for classification. Pre-processing is different for categorical and numerical variables. In order to implement different pre-pricessing flows, I must first classify all of the variables as categorical or numerical. The function below separates columns into these two classes and excludes any variables that will be dropped from the model.

[ ]:
```
def classify_columns(df, drop_cols):
    """Takes a dataframe and a list of columns to
    drop and returns:
        - cat_cols: A list of categorical columns.
        - num_cols: A list of numerical columns.
    """
    cols = df.columns
    keep_cols = [col for col in cols if col not in drop_cols]
    cat_cols = []
    num_cols = []
    for col in keep_cols:
        if df[col].dtype == object:
            cat_cols.append(col)
        else:
            num_cols.append(col)
    return cat_cols, num_cols
```

[ ]:
```
cat_cols, num_cols = classify_columns(X_train, drop_cols)
```

0.0.1 Build Preprocessor

Below I build a preprocessing step for the pipeline which handles all data processing.

Categorical Preprocessing Pipeline The pipeline below executes the following three steps for all of the categorical data. 1. Convert all values in categorical columns to strings. This avoids data type errors in the following steps. 2. Fill all missing values with the string `missing`. 3. One-hot encode all categorical variables. Because this data contains categorical variables with many possible values, it is possible to encounter values in testing data that was not present in the training data. For this reason, I need to set `handel_unknown` to `ignore` so that the encoder will simply ignore unknown values in testing data.

[ ]:
```
categorical_pipeline = Pipeline(
    steps=[
        (
            'typeConverter',
            CategoricalTypeConverter
        ),
        (
            'imputer',
            SimpleImputer(
                strategy='constant',
                fill_value='missing'
            )
        ),
        (
            'standardizer',
            OneHotEncoder(
                handle_unknown='ignore',
                dtype=float
            )
        )
    ]
)
```

Numerical Preprocessing Pipeline The pipeline below executes two steps: 1. Imputes missing values in any numerical column with the median value from that column. 2. Scales each variable to have mean zero and standard deviation one.

[ ]:
```
numerical_pipeline = Pipeline(
    steps=[
        (
            'imputer',
            SimpleImputer(
                strategy='median'
            )
        ),
        (
            'standardizer',
            StandardScaler()
        )
    ]
)
```

Preprocessing Pipeline The column transformer below implements each of the three possible pre-processing behaviors. 1. Apply the categorical pipeline. 2. Apply the numerical pipeline. 3. Drop the specified columns. The if-then statement below ensures that the drop processor is only implemented if there are columns to drop. This is needed since passing an empty `drop_col` list throws an error.

[ ]:
```
if len(drop_cols) > 0:
    preprocessor = ColumnTransformer(
        transformers=[
            (
                'numericalPreprocessor',
                numerical_pipeline,
                num_cols
            ),
            (
                'categoricalPreprocessor',
                categorical_pipeline,
                cat_cols
            ),
            (
                'dropPreprocessor',
                'drop',
                drop_cols
            )
        ]
    )
else:
    preprocessor = ColumnTransformer(
        transformers=[
            (
                'numericalPreprocessor',
                numerical_pipeline,
                num_cols
            ),
            (
                'categoricalPreprocessor',
                categorical_pipeline,
                cat_cols
            )
        ]
    )
```

0.0.2 Build Model Pipeline

Below I build the main pipeline which executes two steps. 1. Apply preprocessing to the raw data. 2. Fit a one vs rest classifier to the processed data using an eXtreme Gradient Boosted forest model.

[ ]:
```
pipeline = Pipeline(
    steps=[
        (
            'preprocessor',
            preprocessor
        ),
        (
            'classifier',
            OneVsRestClassifier(
                estimator='passthrough'
            )
        )
    ]
)
```

0.0.3 Building Parameter Grid

Below I define a grid of hyper-parameters for the pipeline that will be tested in a grid search below.

[ ]:
```
parameter_grid = [
    {
        'classifier__estimator': [
            XGBClassifier()
        ],
        'classifier__estimator__max_depth': [
            5, 10, 15, 20
        ],
        'classifier__estimator__n_estimators': [
            100, 150, 200, 250
        ]
    }
]
```

0.0.4 Instantiate Grid Search

Below I instantiate a grid search object which will fit the pipeline for every combination of the parameters defined above. Since the competition uses accuracy as it's measure of model quality, I sill evaluate model performance in terms of accuracy. For each parameter combination, the grid search will also execute five-fold cross validation.

In order to maximize performance, I will fit the grid search on the full provided training data set and select the best hyper-parameters based on the results of cross validation. For the purposes of local model evaluation, I will then refit the best model on the local training data and use the local testing data to produce a confusion matrix.

[ ]:
```
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=parameter_grid,
    scoring='accuracy',
    cv=5,
    verbose=2,
    n_jobs=-2,
    refit=True
)
```

0.0.5 Fit Grid Search

Below I fit the grid search on the full training set and select the best model hyper-parameters. This step takes an Extremely long time to run.

```
[ ]: grid_search.fit(
         X, y
     )
     model = grid_search.best_estimator_
```

0.0.6 Display Results of Grid Search

Below I display the results of the grid search. I pay particular attention to `std_test_score` which will become larger if the model is over-fit.

```
[ ]: grid_search_results = pd.DataFrame(
         grid_search.cv_results_
     )
     grid_search_results.to_csv(
         '../reports/grid_search_results.csv'
     )
```

```
[ ]: grid_search_results
```

Plotting Model Accuracy

```
[ ]: fig, ax = plt.subplots()
     fig.set_figheight(6)
     fig.set_figwidth(8)
     sns.lineplot(
         x='param_classifier__estimator__max_depth',
         y='mean_test_score',
         hue='param_classifier__estimator__n_estimators',
         data=grid_search_results,
         ax=ax
     )
     handles, labels = ax.get_legend_handles_labels()
     ax.legend(
         handles=handles[1:],
         labels=labels[1:],
         title="Number of Estimators"
     );
     ax.set_xlabel(
         'Max Depth'
     );
     ax.set_ylabel(
         'Mean Test Score'
     );
```

```
ax.set_title(
    'XGBClassifier Model Accuracy'
);
fig.savefig(
    '../images/Model_Accuracy.png',
    bbox_inches='tight'
)
```

Plotting Fit Time

```
[ ]: fig, ax = plt.subplots()
fig.set_figheight(6)
fig.set_figwidth(8)
sns.lineplot(
    x='param_classifier__estimator__max_depth',
    y='mean_fit_time',
    hue='param_classifier__estimator__n_estimators',
    data=grid_search_results,
    ax=ax
)
handles, labels = ax.get_legend_handles_labels()
ax.legend(
    handles=handles[1:],
    labels=labels[1:],
    title="Number of Estimators"
);
ax.set_xlabel(
    'Max Depth'
);
ax.set_ylabel(
    'Mean Fit Time (sec)'
);
ax.set_title(
    'XGBClassifier Model Fit Time'
);
fig.savefig(
    '../images/Model_Fit_Time.png',
    bbox_inches='tight'
)
```

0.0.7 Predict on Validation Data

Below I import the testing data provided by the competition. To maximize performance I refit the model on the full training data set. Predictions are formatted and saved to CSV for submission.

```
[ ]: X_validate = pd.read_csv(
    '../data/testing_features.csv',
    index_col='id'
)
y_validate = model.predict(
    X_validate
)
df_predictions = pd.DataFrame(
    y_validate,
    index=X_validate.index,
    columns=['status_group']
)
df_predictions.to_csv(
    '../predictions/final_model.csv'
)
```

0.0.8 Produce Confusion Matrix

Below I fit the model on the local training data and produce a confusion matrix using the local test data. This provides a reasonable indication of how the model performs. Because the model needs to be fit before producing the matrix, this step will take a long time to run.

Refitting Model on Local Training Set

[ ]:
```
model.fit(
    X_train,
    y_train
)
```

Computing Confusion Matrix on Local Testing Set

[ ]:
```
fig, ax = plt.subplots()
fig.set_figheight(8)
fig.set_figwidth(8)
plot_confusion_matrix(
    model, X_test,
    y_test,
    ax=ax,
    normalize='true',
    include_values=True
)
fig.savefig(
    '../images/Confusion_Matrix.png',
    bbox_inches='tight'
)
```

0.0.9 Confusion Matrix Analysis

I can see that accuracy is worst on the 'functional needs repair' group as I would expect. This class is under represented in the data and is most likely somewhat ambiguously defined in comparison to the other classifications.

0.0.10 Future Work

The two most promising directions for further work seem to be: 1. Integrating re-sampling into the pipeline to improve accuracy on the ‘functional needs repair’ class. 2. Implementing hierarchical models or stacked models.