Python Code for RBIP Model

Alfi Gözaçan[*]

February 9, 2022

[*]Humberside Fire and Rescue Service

Dataset Name	Dimensions	Description	Location
`certificates_ERY.csv`	(5140, 40)	EPC data for East Riding	Click here
`certificates_Hull.csv`	(5576, 40)	EPC data for Hull	Click here
`certificates_NLincs.csv`	(2669, 40)	EPC data for N. Lincs	Click here
`certificates_NELincs.csv`	(2843, 40)	EPC data for N. E. Lincs	Click here
`rbip_df`	(28547, 7)	RBIP data for Humberside	HQCFRMISSQL
`inc_df`	(26820, 5)	Commercial incidents in Humberside	HQIRS

Table 1: Table of datasets used.

1 Overview

The code used in this project can be split into four parts: data cleaning, model training, diagnostics, and interpretation. It is recommended to first read through the report that explains the chosen methods in more detail before reading the code (which can be found here). The following imports are required for the model code. Also note that before executing the code, your machine must have access to the HQCFRMISSQL and HQIRS databases, which might require connecting to the VPN.

```
import numpy as np
import pandas as pd
import pyodbc
import scikitplot as skplt
import matplotlib.pyplot as plt
import seaborn as sns

from fuzzywuzzy import fuzz
from tqdm import tqdm
from scipy import stats
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import RandomOverSampler
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
from sklearn.impute import KNNImputer
```

2 Data Cleaning

Table 1 summarises the data collected and used in this non-domestic risk model. Each dataset is either available in `.csv` format or stored on a SQL server that can be connected to via ODBC (which will be accounted for in the code).

Firstly, the EPC data is imported in its individual local authorities, e.g. `certificates_ERY.csv` contains the energy performance certificates of commercial properties in the East Riding of Yorkshire. Then all of these dataframes are combined into one pandas DataFrame object.

```
file_path = "C:\\path_to_data\\"
loc_auths = ["ERY", "Hull", "NLincs", "NELincs"]

for i in range(len(loc_auths)):
    if i == 0:
        epc_df = pd.read_csv(file_path + "certificates_" + loc_auths[i] + ".csv")
    else:
        epc_df = pd.concat([epc_df, pd.read_csv(file_path +
                                                "certificates_" +
                                                loc_auths[i] +
                                                ".csv")]).reset_index(drop=True)

epc_df.drop(epc_df[epc_df["ASSET_RATING_BAND"].isin(["A+", "INVALID!"])].index,
            axis=0,
            inplace=True)
epc_df.reset_index(drop=True, inplace=True)
epc_df.replace(np.nan, "", inplace=True)
```

Then, the gazetteer data is imported from the CFRMIS_HUMBS SQL server using pyodbc.

```
server = "HQCFRMISSQL"
database = "CFRMIS_HUMBS"
cnxn = pyodbc.connect("DRIVER={SQL Server};SERVER="+server+";DATABASE="+database)

query = '''
select *
from ADDRESS_GAZ
where PREMISE_DESCRIPTION like 'Commercial%'
'''

address_df = pd.read_sql(query, cnxn)
```

Next, the address dataframe is used to attach UPRNs to the records in the EPC data where a UPRN is not currently present.

```
address_strings = []

for i in range(len(address_df)):
    string = " ".join(entry for entry in address_df.iloc[i, 5:10])
    address_strings.append(string)

epc_strings = []

for i in epc_df[epc_df["UPRN"] == ""].index:
    string = " ".join(entry for entry in epc_df.iloc[i, 2:5])
    epc_strings.append(string)

matching_indices = []

for i in tqdm(range(len(epc_strings))):
    fuzz_ratios = []
    for j in range(len(address_strings)):
        if epc_strings[i][-8:] == address_strings[j][-8:]:
            fuzz_ratios.append(fuzz.token_set_ratio(epc_strings[i], address_strings[j]))
        else:
            fuzz_ratios.append(0)
    index = fuzz_ratios.index(max(fuzz_ratios))
    matching_indices.append(index)
```

```
epc_df.loc[epc_df[epc_df["UPRN"] == ""].index, "UPRN"] = list(address_df.iloc[matching_indices, 1])
```

Next, any duplicate UPRNs are combined into a single entry by averaging over the numerical values and taking the mode of the ordinal values.

```
duplicate_UPRNs = epc_df["UPRN"].value_counts().rename_axis("UPRN").reset_index(name="COUNT")
duplicate_UPRNs.drop(duplicate_UPRNs.index[duplicate_UPRNs["COUNT"] == 1], axis=0, inplace=True)
print("\n")

for i in tqdm(range(len(duplicate_UPRNs))):
    sub_df = epc_df[epc_df["UPRN"] == duplicate_UPRNs.iloc[i,0]]
    new_entry = sub_df.iloc[0,:].copy()
    new_entry["ASSET_RATING"] = np.mean(sub_df["ASSET_RATING"])
    new_entry["ASSET_RATING_BAND"] = stats.mode(sub_df["ASSET_RATING_BAND"])[0][0]
    epc_df.drop(sub_df.index, axis=0, inplace=True)
    epc_df.loc[sub_df.index[0]] = new_entry

epc_df.reset_index(drop=True, inplace=True)
```

Then, the RBIP data is imported from the CFRMIS_HUMBS SQL server.

```
query = '''
select UPRN,
       CUSTODIAN,
       RISK_OF_FIRE,
       SEVERITY_OF_FIRE,
       SLEEPING_RISK,
       SLEEPING_RISK_ABOVE,
       SLEEPING_RISK_SCORE,
       SSRI_SCORE
from RBIP_2021
where RISK_OF_FIRE != 'NULL'
and SEVERITY_OF_FIRE != 'NULL'
and SLEEPING_RISK != 'NULL'
and SLEEPING_RISK_ABOVE != 'NULL'
and SLEEPING_RISK_SCORE != 'NULL'
and SSRI_SCORE != 'NULL'
'''

rbip_df = pd.read_sql(query, cnxn)
```

After that, both the EPC dataframe and the RBIP dataframe are merged into one, joined on UPRN in an *outer* join (meaning there will be some NAs).

```
epc_df["UPRN"] = [int(uprn) for uprn in epc_df["UPRN"]]
rbip_df["UPRN"] = [int(uprn) for uprn in rbip_df["UPRN"]]
rbip_epc_df = rbip_df.merge(right=epc_df, left_on="UPRN", right_on="UPRN", how="outer")
```

Then, the incidents data is loaded from the threetc_irs server.

```
server = "HQIRS"
database = "threetc_irs"
cnxn = pyodbc.connect("DRIVER={SQL Server};SERVER="+server+";DATABASE="+database)
```

```
query = '''
select  V_VISION_INCIDENT.REVISED_INCIDENT_TYPE,
        V_VISION_INCIDENT.GAZETTEER_URN,
        V_VISION_INCIDENT.CREATION_DATE,
        inc_incident.inc_location_address,
        inc_incident.inc_property_type
from V_VISION_INCIDENT
join inc_incident
on V_VISION_INCIDENT.INCIDENT_NUMBER = inc_incident.inc_incident_ref
where isnumeric(GAZETTEER_URN) = 1
and inc_incident.inc_property_type in (
    select CODE
    from Reporting_MENU_IRS_PROPERTY_TYPE
    where CATEGORY = 'Commercial'
)
'''

inc_df = pd.read_sql(query, cnxn)
```

Next, the incidents are cross-tabulated by year and the dataframe is merged onto the RBIP and EPC dataframe using UPRN in a left join (as some properties may not have had an incident).

```
inc_df["GAZETTEER_URN"] = [int(uprn) for uprn in inc_df["GAZETTEER_URN"]]
inc_df["YEAR"] = ["inc." + str(inc_df.loc[i, "CREATION_DATE"])[:4] for i in range(len(inc_df))]
inc_crosstab = pd.crosstab(inc_df["GAZETTEER_URN"],
                           inc_df["YEAR"]).rename_axis("UPRN").reset_index()
rbip_epc_inc_df = rbip_epc_df.merge(right=inc_crosstab, left_on="UPRN", right_on="UPRN", how="left")
```

Then, the columns to be used in the machine learning algorithms are selected. Note that although UPRN is selected, it will not be used in the model but in the output.

```
model_cols = ["UPRN",
              "CUSTODIAN",
              "RISK_OF_FIRE",
              "SEVERITY_OF_FIRE",
              "SLEEPING_RISK",
              "SLEEPING_RISK_ABOVE",
              "SSRI_SCORE",
              "ASSET_RATING",
              "ASSET_RATING_BAND",
              "PROPERTY_TYPE",
              "MAIN_HEATING_FUEL",
              "FLOOR_AREA",
              "BUILDING_EMISSIONS",
              "PRIMARY_ENERGY_VALUE",
              "inc.2010",
              "inc.2011",
              "inc.2012",
              "inc.2013",
              "inc.2014",
              "inc.2015",
              "inc.2016",
              "inc.2017",
              "inc.2018",
             "inc.2019",
             "inc.2020"]

rbip_epc_inc_df["ASSET_RATING_BAND"] = rbip_epc_inc_df["ASSET_RATING_BAND"].replace(["A",
                                                                                      "B",
                                                                                      "C",
                                                                                      "D",
                                                                                      "E",
                                                                                      "F",
                                                                                      "G"], range(7))

rbip_epc_inc_df["CUSTODIAN"] = rbip_epc_inc_df["CUSTODIAN"].replace(
                                                    ["Crew",
                                                     "Protection Team"],
                                                    [0, 1])

categorical_cols = ["SLEEPING_RISK",
                    "PROPERTY_TYPE",
                    "MAIN_HEATING_FUEL"]

numerical_cols = ["RISK_OF_FIRE",
                  "SEVERITY_OF_FIRE",
                  "SLEEPING_RISK_ABOVE",
                  "SSRI_SCORE",
                  "ASSET_RATING",
                  "ASSET_RATING_BAND",
                  "FLOOR_AREA",
                  "BUILDING_EMISSIONS",
                  "PRIMARY_ENERGY_VALUE"]

rbip_epc_inc_df = rbip_epc_inc_df[model_cols]
rbip_epc_inc_df.replace("", 0, inplace=True)
```

Next, a copy of this combined dataframe is saved in order to be used later for training purposes. Then, any missing data is imputed on the original dataframe by taking appropriate averages across all the existing data, using nearest neighbours. Note that premises with missing CUSTODIAN are assigned to crews (labelled 0).

```
model_df = rbip_epc_inc_df.copy()
model_df.dropna(axis=0, subset=numerical_cols, inplace=True)
model_df.replace(np.nan, 0, inplace=True)
model_df.reset_index(drop=True, inplace=True)

impute_df = rbip_epc_inc_df.copy()

for j in range(len(categorical_cols)):
    null_indices = impute_df.index[impute_df[categorical_cols[j]].isna()]
    non_null_indices = impute_df.index[~impute_df[categorical_cols[j]].isna()]
    mode = stats.mode(impute_df.loc[non_null_indices, categorical_cols[j]])[0][0]
    impute_df.loc[null_indices, categorical_cols[j]] = mode

null_indices = impute_df.index[impute_df["CUSTODIAN"].isna()]
impute_df.loc[null_indices, "CUSTODIAN"] = 0
```

```
imputer = KNNImputer(weights="distance")
impute_df.loc[:,numerical_cols] = imputer.fit_transform(impute_df[numerical_cols])
impute_df.replace(np.nan, 0, inplace=True)
```

Next, the final column is transformed into a binary response variable denoting whether an incident did or did not occur in the year 2020. The exact same is done for `impute_df`.

```
model_df.loc[model_df[model_df["inc.2020"] > 0].index, "inc.2020"] = 1
model_df.rename({"inc.2020": "inc.2020.bool"}, axis=1, inplace=True)
```

Finally, the combined dataframe is dummy encoded so that it is ready for the classifier algorithms. The exact same is done for `impute_df`. Then, some columns that arise in `impute_df` that are not present in `model_df` are removed.

```
encoder = OneHotEncoder(drop="first", sparse=False)
dummy_view = encoder.fit_transform(model_df[categorical_cols])
encoded_model_df = pd.DataFrame(dummy_view)
encoded_model_df.columns = encoder.get_feature_names(categorical_cols)
model_df.drop(categorical_cols, axis=1, inplace=True)
model_df = encoded_model_df.join(model_df)

cols = model_df.columns.tolist()
cols.remove("UPRN")
cols.insert(0, "UPRN")
model_df = model_df[cols]

impute_df.loc[impute_df[impute_df["inc.2020"] > 0].index, "inc.2020"] = 1
impute_df.rename({"inc.2020": "inc.2020.bool"}, axis=1, inplace=True)
encoder = OneHotEncoder(drop="first", sparse=False)
dummy_view = encoder.fit_transform(impute_df[categorical_cols])
encoded_impute_df = pd.DataFrame(dummy_view)
encoded_impute_df.columns = encoder.get_feature_names(categorical_cols)
impute_df.drop(categorical_cols, axis=1, inplace=True)
impute_df = encoded_impute_df.join(impute_df)

cols = impute_df.columns.tolist()
cols.remove("UPRN")
cols.remove("CUSTODIAN")
cols.insert(0, "CUSTODIAN")
cols.insert(0, "UPRN")
impute_df = impute_df[cols]

surplus_columns = [col for col in impute_df.columns if col not in model_df.columns]
impute_df.drop(surplus_columns, axis=1, inplace=True)
```

3 Model Training

Firstly, the dataframe is split into training and testing data in the ratio 67%-33%.

```
training_set, test_set = train_test_split(model_df, test_size = 0.33, random_state=1)
```

Then, the minority class of the training data is randomly oversampled to give a balanced dataset, before the training and testing sets are reassigned into their covariate matrix and response components. Note that the covariate matrices ignore the first two columns, `UPRN` and `CUSTODIAN`.

```
oversamp = RandomOverSampler(random_state=1)
X, y = oversamp.fit_resample(training_set.iloc[:,:-1], training_set.iloc[:,-1])
training_set = pd.DataFrame(X)
training_set["inc.2020.bool"] = y

X_train = training_set.iloc[:,2:-1]
y_train = training_set.iloc[:,-1]
X_test = test_set.iloc[:,2:-1]
y_test = test_set.iloc[:,-1]
```

Now, the supervised machine learning algorithms are trained on the data. Note the random state is set to 1 for reproducibility.

```
adaboost = AdaBoostClassifier(random_state=1)
adaboost.fit(X_train, y_train)

rf = RandomForestClassifier(random_state=1)
rf.fit(X_train, y_train)

logreg = LogisticRegression(random_state=1, solver="liblinear")
logreg.fit(X_train, y_train)

xgboost = GradientBoostingClassifier(random_state=1)
xgboost.fit(X_train, y_train)
```

4 Model Evaluation

Firstly, the model predictions are appended to the test dataset.

```
y_ada_pred = adaboost.predict(X_test)
test_set.insert(ncols, "AdaBoost Predictions", y_ada_pred)

y_rf_pred = (rf.predict_proba(X_test)[:,1] >= 0.3).astype(float)
test_set.insert(ncols+1, "RF Predictions", y_rf_pred)

y_lr_pred = logreg.predict(X_test)
test_set.insert(ncols+2, "LogReg Predictions", y_lr_pred)

y_xg_pred = xgboost.predict(X_test)
test_set.insert(ncols+3, "XGBoost Predictions", y_xg_pred)
```

Then, the number of positive assignments are calculated and printed, before confusion matrices are outputted with precision and recall.

```
real_positives = len(test_set[test_set["inc.2020.bool"] == 1.0])
adaboost_positives = len(test_set[test_set["AdaBoost Predictions"] == 1.0])
rf_positives = len(test_set[test_set["RF Predictions"] == 1.0])
logreg_positives = len(test_set[test_set["LogReg Predictions"] == 1.0])
XGBoost_positives = len(test_set[test_set["XGBoost Predictions"] == 1.0])

print(f'''There are {len(test_set)} entries in the test set,
       of which {real_positives} are real positives''')
print(f"AdaBoost predicted {adaboost_positives} positives")
print(f"Random Forest predicted {rf_positives} positives")
print(f"Logistic Regression predicted {logreg_positives} positives")
print(f"XGBoost predicted {XGBoost_positives} positives")

print("AdaBoost:\n", classification_report(test_set.iloc[:,ncols-1],
                                           test_set.iloc[:,ncols]))
print("Random Forest:\n", classification_report(test_set.iloc[:,ncols-1],
                                                test_set.iloc[:,ncols+1]))
print("Logistic Regression:\n", classification_report(test_set.iloc[:,ncols-1],
                                                      test_set.iloc[:,ncols+2]))
print("XGBoost:\n", classification_report(test_set.iloc[:,ncols-1],
                                          test_set.iloc[:,ncols+3]))
```

The proportion of records correctly guessed as either having no incidents in 2020 or having an incident in 2020 is printed next for each model.

```
length = len(test_set.iloc[:,ncols-1])

ada_no_matched = sum([(test_set.iloc[i,ncols-1] * test_set.iloc[i,ncols]) +
                      ((1-test_set.iloc[i,ncols-1]) * (1-test_set.iloc[i,ncols]))
                      for i in range(length)])
rf_no_matched = sum([(test_set.iloc[i,ncols-1] * test_set.iloc[i,ncols+1]) +
                     ((1-test_set.iloc[i,ncols-1]) * (1-test_set.iloc[i,ncols+1]))
                     for i in range(length)])
lr_no_matched = sum([(test_set.iloc[i,ncols-1] * test_set.iloc[i,ncols+2]) +
                     ((1-test_set.iloc[i,ncols-1]) * (1-test_set.iloc[i,ncols+2]))
                     for i in range(length)])
xg_no_matched = sum([(test_set.iloc[i,ncols-1] * test_set.iloc[i,ncols+3]) +
                     ((1-test_set.iloc[i,ncols-1]) * (1-test_set.iloc[i,ncols+3]))
                     for i in range(length)])

ada_accuracy = ada_no_matched / length
rf_accuracy = rf_no_matched / length
lr_accuracy = lr_no_matched / length
xg_accuracy = xg_no_matched / length

print("AdaBoost Proportion Correctly Guessed:", ada_accuracy)
print("Random Forest Proportion Correctly Guessed:", rf_accuracy)
print("Logistic Regression Proportion Correctly Guessed:", lr_accuracy)
print("XGBoost Proportion Correctly Guessed:", xg_accuracy)
```

5 Model Output

To produce a prioritised list of commercial premises, we simply rename the probabilities given by the model as “risk score”, before splitting the results into their custodian classes (A = Protection Team, B = Crew). Then the two streams are quartiled and the whole output is saved. Note that some random noise must be added to the probabilities in order for there to be no duplicates, which allows for the quartiling to work properly.

```
all_adaprobs = adaboost.predict_proba(impute_df.iloc[:,2:-1])
adaprobs = adaboost.predict_proba(X_test)
rfprobs = rf.predict_proba(X_test)
lrprobs = logreg.predict_proba(X_test)
```

```
xgprobs = xgboost.predict_proba(X_test)

positive_probs = (np.array([x[1] for x in all_adaprobs]) +
                  np.random.random(size=len(impute_df)) * 1e-5)
complex_indices = impute_df.index[impute_df["CUSTODIAN"] == 1]
non_complex_indices = impute_df.index[impute_df["CUSTODIAN"] == 0]
complex_positive_probs = [positive_probs[i] for i in complex_indices]
non_complex_positive_probs = [positive_probs[i] for i in non_complex_indices]
impute_df.loc[complex_indices, "QUARTILE"] = pd.qcut(complex_positive_probs,
                                                     q=4,
                                                     labels=[4, 3, 2, 1])
impute_df.loc[non_complex_indices, "QUARTILE"] = pd.qcut(non_complex_positive_probs,
                                                         q=4,
                                                         labels=[4, 3, 2, 1])
impute_df.to_csv(file_path + "output.csv", index=False)
```

6 Plots

This code produces ROC curves and a feature importance graph, showing the most indicative factors of the data from the perspective of the model.

```
probas = [adaprobs, rfprobs, lrprobs, xgprobs]
titles = ["AdaBoost", "Random Forest", "Logistic Regression", "XGBoost"]

for i in range(len(probas)):
    skplt.metrics.plot_roc(y_test, probas[i], title=titles[i])
    plt.savefig(file_path+"roc_"+str(i)+".png", dpi = 200, bbox_inches = "tight")

features = rf.feature_importances_
ftrs = pd.DataFrame({"column_name": model_df.columns[2:-1],
                     "score": features}).sort_values(
                                        by="score",
                                        ascending=False).reset_index(drop=True)

ftrs.to_csv(file_path + "feature_importance_scores.csv", index=False)

plt.figure(figsize=(10,8))
sns.barplot(y = ftrs.loc[:30, "column_name"], x = ftrs.loc[:30, "score"])
plt.title("Random Forest Feature Importance")
plt.xlabel("Score")
plt.ylabel("Feature Names")
plt.savefig(file_path+"feature_importance.png", dpi = 200, bbox_inches = "tight")
```