Bankruptcy prediction

In the following we wil go through the code for this project. We will start with the Data Cleaning process.

Data Preparer Class

Responsible for feature engineering, imputation, generation and selection.

```
class DataPreparer:


    def __init__(self, feature_select='tree'):
        self.df = self.load_data()

    def load_data(self):
        #df1 = pd.read_csv('data/1year.csv')
        #df1['time'] = 5
        #df2 = pd.read_csv('data/2year.csv')
        #df2['time'] = 4
        df3 = pd.read_csv('data/3year.csv')
        df3['time'] = 3
        df4 = pd.read_csv('data/4year.csv')
        df4['time'] = 2
        df5 = pd.read_csv('data/5year.csv')
        df5['time'] = 1
        df = pd.concat([df3, df4, df5])
        df['class'] = pd.to_numeric(df['class'].apply(lambda x: re.search('\d+', x).group(0)))
        df['is_bankrupt_after_years'] = df['class'] * df['time']
        df = df.drop(['class', 'time'], axis=1)
        return df
```

We create a class which loads our dataset. We only use the last three years for prediction but can easily extend it to the full five year prediction.

Thereafter we impute the missing values of the dataset with the simple imputer from sklearn and choose the mean value as the imputation strategy.

As the dataset is strongly imbalenced which would restrict our training process, we use the Synthetic Minority Oversampling Technique - short SMOTE. Thereby we create aritifcial bankrupt cases for 1 year, 2 years and 3 years.

As we do have a lot of features, which describe the same (which can be seen from the Kendall correlation heatmap matrix) we need to select the features with the highest influence on our target variable. We tested two different functions. An ExtraTreesClassifier, which uses a random forest to select the highest influencing Features and a KBestSelection which uses statistical metrics to select the most influential variables. Our setting uses the ANOVA feature selection. With the factor K we are able to select the k best features.

At the end of each process we save the modified data to the classes dataframe

```
     def impute_missing_values(self, showBeforeAndAfter=False):
        self.df = pd.DataFrame(SimpleImputer(missing_values=np.nan, strategy='mean').fit_transform(sel
                               columns=self.df.columns)

    def synthetic_minority_oversampling(self):
        oversample = BorderlineSMOTE()
        X = self.df.drop('is_bankrupt_after_years', axis=1)
        y = self.df['is_bankrupt_after_years']
        X_new, y_new = oversample.fit_resample(X, y)
        X_new['is_bankrupt_after_years'] = y_new
        self.df = X_new

    def feature_selection(self, feature_select):
        X = self.df.drop('is_bankrupt_after_years', axis=1)
        y = self.df.is_bankrupt_after_years
        if feature_select == 'tree':
            from sklearn.ensemble import ExtraTreesClassifier
            from sklearn.feature_selection import SelectFromModel
            clf = ExtraTreesClassifier(n_estimators=50)
            clf = clf.fit(X, y)
            model = SelectFromModel(clf, prefit=True)
            dfNew = pd.DataFrame(model.transform(X), columns=X.columns[model.get_support()])

        if feature_select == 'kbest':
            # ANOVA feature selection for numeric input and categorical output
            from sklearn.feature_selection import SelectKBest
            from sklearn.feature_selection import f_classif
            fs = SelectKBest(score_func=f_classif, k=20)
            X_selected = fs.fit_transform(X, y)
            dfNew = pd.DataFrame(X_selected, columns=X.columns[fs.get_support()])
        dfNew['is_bankrupt_after_years'] = y
        self.df = dfNew

    def min_max_scale(self):
        X = self.df.drop('is_bankrupt_after_years', axis=1)
        min_max_scaler = preprocessing.MinMaxScaler()
        x_scaled = min_max_scaler.fit_transform(X)
        df = pd.DataFrame(x_scaled)
        df['is_bankrupt_after_years'] = self.df['is_bankrupt_after_years']
        self.df = df
```

The Model

In this class we create a Model instance first with the requested model and train test split ratio. With predict() we are able to train different models based on the beforehand model selection. We are able to save our model to load it for later Model Serving

```
class Model:

    def __init__(self, df, model='RandomForest', test_split=0.2):
        self.df = df
        self.model = model
        X = self.df.drop('is_bankrupt_after_years', axis=1)
        y = self.df['is_bankrupt_after_years']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X, y, test_size=test_s
                                                                                 random_state=0)
        self.predict()

    def predict(self):
        if self.model == 'DecisionTree':  # 0.9084627799072019
            clf = tree.DecisionTreeClassifier()
            clf = clf.fit(self.X_train, self.y_train)
            self.y_pred = clf.predict(self.X_test)
            self.clf = clf
        if self.model == 'RandomForest':  # 0.9807343151099456
            clf = RandomForestClassifier(max_depth=100, random_state=0)
            clf = clf.fit(self.X_train, self.y_train)
            #self.y_pred = clf.predict_proba(self.X_test)
            self.y_pred = clf.predict(self.X_test)
            self.classifier = clf
        if self.model == 'AdaBoost':  # 0.49399838612063746
            clf = AdaBoostClassifier(n_estimators=100, random_state=0)
            clf = clf.fit(self.X_train, self.y_train)
            self.y_pred = clf.predict(self.X_test)
        if self.model == 'KNeighbors':  # 0.7816219487593302
            clf = KNeighborsClassifier(n_neighbors=3)
            clf = clf.fit(self.X_train, self.y_train)
            self.y_pred = clf.predict(self.X_test)
        if self.model == 'XGBoost':  # 0.9425055477103087
            model = XGBClassifier(eval_metric="mlogloss")
            model.fit(self.X_train, self.y_train)
            self.y_pred = model.predict_proba(self.X_test)
            self.y_pred = [np.argmax(ele) for ele in self.y_pred]
            self.classifier = model

    def save_model_for_serving(self):
        joblib.dump(self.classifier, 'Bankruptcy_' + self.model + '.joblib')
```

Exploratory Data Analysis

The exploratory data analysis class contains methods which are used to create graphs to visualize our data for the presentation and to gain further insight into the data.

```
class EDA:

    def __init__(self, df):
        self.df = df

    def distribution(self, object='is_bankrupt_after_years', turnLabel=False, title='Title', barplot=Tr
        df = self.df
        if turnLabel:
            plt.xticks(rotation=90)
        plt.title(title, fontsize=18, color='black')
        if barplot:
            sns.countplot(x=df[object])
        else:
            df.groupby(object).size().plot(kind='pie', autopct='%1.1f%%', pctdistance=0.75)
            plt.ylabel("")
            plt.legend(['0: 24,785', '1: 410', '2: 515', '3: 495'], loc=2)
            plt.legend(['0: 24,785', '1: 24,785', '2: 24,785', '3: 24,785'], loc=2)
        #plt.show()
        plt.savefig(path, bbox_inches='tight')
```

Model serving

For model serving we use the python MLServing package. It starts a local webserver which is able to process post requests. With the files settings.json and model-settings.json, we are able to select a model which we want to use. We earlier saved a file in the model class. Follwing is a function to test the service and send a REST request

```
def request_to_model():
    df = pd.read_csv('data/cleandata.csv')
    X = df.drop('is_bankrupt_after_years', axis=1)
    y = df['is_bankrupt_after_years']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.0002, stratify=y)
    inference_request = {'inputs': [{'name': 'predict', 'shape': X_test.shape,
                                     'datatype': 'FP64', 'data': X_test.values.tolist()}]}
    endpoint = 'http://localhost:8080/v2/models/bankrupt-sklearn/versions/v1/infer'
    response = requests.post(endpoint, json=inference_request)
    print(response.text)
    print(y_test.values)
```