```python
import DataPackage as dp
import DataExperimentSupport as des
import copy
import pickle


class DataExperiment:

    def __init__(self,
                 projectName,
                 experimentName,
                 origData,
                 uniqueColumn,
                 targetColumn,
                 classifier,
                 processDataPackage=False
                 ):
        self.projectName = projectName
        self.experimentName = experimentName
        self.__setDataPackage(origData=origData,
                              uniqueColumn=uniqueColumn,
                              targetColumn=targetColumn)
        self.__setClassifier(classifier)

        # Should really consider putting these into a function
        # Following are default values on init for stuff set later
        self.isBaseModelLoaded = False
        self.baseModel = None
        self.isFinalModelLoaded = False
        self.finalModel = None

        self.isBaseModelPredicted = False
        self.baseModelPrediction = None
        self.baseModelAccuracy = None
        self.baseModelPrecision = None
        self.baseModelRecall = None
        self.baseModelF1 = None
        self.baseModelCohenKappa = None

        self.isFinalModelPredicted = False
        self.finalModelPrediction = None
        self.finalModelPrediction = None
        self.finalModelAccuracy = None
        self.finalModelPrecision = None
        self.finalModelRecall = None
        self.finalModelF1 = None
        self.finalModelCohenKappa = None

        self.isBaseModelLearningCurveCreated = False
        self.baseModel_train_sizes = None
        self.baseModel_train_scores = None
        self.baseModel_test_scores = None
        self.baseModel_fit_times = None
        self.baseModelROCAUC = None

        self.isFinalModelLearningCurveCreated = False
        self.finalModel_train_sizes = None
        self.finalModel_train_scores = None
        self.finalModel_test_scores = None
        self.finalModel_fit_times = None

        self.isBaseModelROCAUCCalculated = False
        self.baseModelROCAUC = None
        self.isFinalModelROCAUCCalculated = False
        self.finalModelROCAUC = None

        self.finalFeaturesAll = None
        self.finalFeatures = None

    # ===============================================

        if processDataPackage:
```

```python
            self.processDataPackage()

        self.display()

    def display(self):
        indent = '---> '
        print(f'DataExperiment summary:')
        print(f'{indent}projectName: {self.projectName}')
        print(f'{indent}experimentName: {self.experimentName}')
        print(f'{indent}isDataPackageLoaded: {self.isDataPackageLoaded}')

        print(f'{indent}isBaseModelLoaded: {self.isBaseModelLoaded}')
        print(f'{indent}isBaseModelPredicted: {self.isBaseModelPredicted}')
        print(f'{indent}isBaseModelLearningCurveCreated: {self.isBaseModelLearningCurveCreated}')

        print(f'{indent}isFinalModelLoaded: {self.isFinalModelLoaded}')
        print(f'{indent}isFinalModelPredicted: {self.isFinalModelPredicted}')
        print(f'{indent}isFinalModelLearningCurveCreated: {self.isFinalModelLearningCurveCreated}')

        print(f'{indent}isClassifierLoaded: {self.isClassifierLoaded}')
        print(self.getClassifier())
        print('')
        self.dataPackage.display()

    def getClassifier(self):
        return copy.deepcopy(self.classifier)

    def __setClassifier(self, classifier):
        self.classifier = classifier
        self.isClassifierLoaded = True

    def __setDataPackage(self,
                         origData,
                         uniqueColumn,
                         targetColumn):

        self.dataPackage = dp.DataPackage(origData=origData,
                                          uniqueColumn=uniqueColumn,
                                          targetColumn=targetColumn)
        self.isDataPackageLoaded = True

    def processDataPackage(self, verbose=False):
        self.dataPackage.classBalanceUndersample()
        self.dataPackage.splitTrainTest()

        if verbose:
            self.dataPackage.display()

    def createBaseModel(self):
        model = des.createModel(data=self.dataPackage.getTrainData(),
                                uniqueColumn=self.dataPackage.uniqueColumn,
                                targetColumn=self.dataPackage.targetColumn,
                                classifier=self.getClassifier())

        self.__setBaseModel(model)

    def createFinalModel(self,
                                featureImportanceThreshold=0.002):
        # TO DO:
        # need to get list of features we wish to use to subset full data

        impFeatureListFull = self.__getFinalModelFeatures(returnAbove=featureImportanceThreshold,
                                                          includeUniqueAndTarget=True)

        impFeatureList = self.__getFinalModelFeatures(returnAbove=featureImportanceThreshold,
                                                      includeUniqueAndTarget=False)

        # get full training dataframe
        df = self.dataPackage.getTrainData()

        model = des.createModel(data=df[impFeatureListFull],
                                uniqueColumn=self.dataPackage.uniqueColumn,
                                targetColumn=self.dataPackage.targetColumn,
                                classifier=self.getClassifier())
        self.__setFinalModel(model=model,
                             finalFeatures=impFeatureList,
                             finalFeaturesAll=impFeatureListFull)

    def __setBaseModel(self, model):
        self.baseModel = model
        self.isBaseModelLoaded = True

        # when you set base model invalidate some things
        self.isBaseModelPredicted = False
        self.baseModelPrediction = None

    def __setFinalModel(self,
                        model,
                        finalFeatures,
                        finalFeaturesAll):
        self.finalModel = model
        self.isFinalModelLoaded = True
        self.finalFeaturesAll = finalFeaturesAll
        self.finalFeatures = finalFeatures

        # when you set base model invalidate some things
        self.isFinalModelPredicted = False
        self.finalModelPrediction = None

    def getBaseModel(self):
        return self.baseModel

    def getFinalModel(self):
        return self.finalModel

    def __setBaseModelPrediction(self,
                                 predictionData,
                                 colActual,
                                 colPredict,
                                 average='weighted',
                                 sigDigs=2):
        self.baseModelPrediction = predictionData
        self.isBaseModelPredicted = True
        self.baseModelPredictionColActual = colActual
        self.baseModelPredictionColPredict = colPredict

        self.baseModelAccuracy = round(des.
getModelAccuracy(data=predictionData,
          colActual=colActual,
          colPredict=colPredict), sigDigs)

        self.baseModelPrecision = round(des.
getModelPrecision(data=predictionData,
            colActual=colActual,
            colPredict=colPredict,
            average=average), sigDigs)

        self.baseModelRecall = round(des.
getModelRecall(data=predictionData,
  colActual=colActual,
  colPredict=colPredict,
  average=average), sigDigs)

        self.baseModelF1 = round(des.getModelF1(data=
predictionData,
colActual=colActual,
colPredict=colPredict,
average=average), sigDigs)

        self.baseModelCohenKappa = round(des.
getModelCohenKappa(data=predictionData,
              colActual=colActual,
              colPredict=colPredict), sigDigs)

    def __setFinalModelPrediction(self,
                                  predictionData,
                                  colActual,
                                  colPredict,
                                  average='weighted',
                                  sigDigs=2):
        self.finalModelPrediction = predictionData
        self.isFinalModelPredicted = True
        self.finalModelPredictionColActual = colActual
        self.finalModelPredictionColPredict = colPredict

        self.finalModelAccuracy = round(des.getModelAccuracy(data=predictionData,
                                                             colActual=colActual,
                                                             colPredict=colPredict), sigDigs)

        self.finalModelPrecision = round(des.getModelPrecision(data=predictionData,
                                                               colActual=colActual,
                                                               colPredict=colPredict,
                                                               average=average), sigDigs)

        self.finalModelRecall = round(des.getModelRecall(data=predictionData,
                                                         colActual=colActual,
                                                         colPredict=colPredict,
                                                         average=average), sigDigs)
```

```python

        self.finalModelF1 = round(des.getModelF1(data=predictionData,
                                                 colActual=colActual,
                                                 colPredict=colPredict,
                                                 average=average), sigDigs)

        self.finalModelCohenKappa = round(des.getModelCohenKappa(data=predictionData,
                                                                 colActual=colActual,
                                                                 colPredict=colPredict), sigDigs)

    def showBaseModelStats(self):
        print(f'Base Model Stats:')
        print(f'Accuracy: {self.baseModelAccuracy}')
        print(f'Precision: {self.baseModelPrecision}')
        print(f'Recalll: {self.baseModelRecall}')
        print(f'F1 Score: {self.baseModelF1}')
        print(f'Cohen kappa:: {self.baseModelCohenKappa}')

    def showFinalModelStats(self):
        print(f'Final Model Stats:')
        print(f'Accuracy: {self.finalModelAccuracy}')
        print(f'Precision: {self.finalModelPrecision}')
        print(f'Recalll: {self.finalModelRecall}')
        print(f'F1 Score: {self.finalModelF1}')
        print(f'Cohen kappa:: {self.finalModelCohenKappa}')

    def getBaseModelPrediction(self):
        if self.isBaseModelPredicted:
            return self.baseModelPrediction
        else:
            print(f'No base model predictions calculated.')
            return None

    def getFinalModelPrediction(self):
        if self.isFinalModelPredicted:
            return self.finalModelPrediction
        else:
            print(f'No final model predictions calculated.')
            return None

    def predictBaseModel(self, average='weighted'):
        if self.isBaseModelPredicted:
            display("Base model already predicted. Displaying results:")
            self.showBaseModelStats()
            return

        tDf, colActual, colPredict = des.predictModel(model=self.getBaseModel(),
                                                      data=self.dataPackage.getTestData(),
                                                      uniqueColumn=self.dataPackage.uniqueColumn,
                                                      targetColumn=self.dataPackage.targetColumn)

        self.__setBaseModelPrediction(predictionData=tDf,
                                      colActual=colActual,
                                      colPredict=colPredict,
                                      average=average)

        self.showBaseModelStats()

    def predictFinalModel(self, average='weighted'):
        if self.isFinalModelPredicted:
            display("Final model already predicted. Displaying results:")
            self.showFinalModelStats()
            return

        testData = self.dataPackage.getTestData()
        testData = testData[self.finalFeaturesAll].copy()

        tDf, colActual, colPredict = des.predictModel(model=self.getFinalModel(),
                                                      data=testData,
                                                      uniqueColumn=self.dataPackage.uniqueColumn,
                                                      targetColumn=self.dataPackage.targetColumn)

        self.__setFinalModelPrediction(predictionData=tDf,
                                       colActual=colActual,
                                       colPredict=colPredict,
                                       average=average)
        self.showFinalModelStats()

    def analyzeBaseModelFeatureImportance(self,
                                          returnAbove=0.002,
                                          startValue=0.0001,
                                          increment=0.
```

```
0001,
                                          upperValue=0.01,
                                          showSummary=True,
                                          showPlot=True):

        df, featureLabel, valueLabel = des.getModelFeatureImportance(self.getBaseModel())

        retDf = des.analyzeModelFeatureImportance(data=df,
                                                  valueLabel=valueLabel,
                                                  startValue=startValue,
                                                  increment=increment,
                                                  upperValue=upperValue,
                                                  returnAbove=returnAbove,
                                                  showSummary=showSummary,
                                                  showPlot=showPlot)
        return retDf

    def analyzeFinalModelFeatureImportance(self,
                                           returnAbove=0.002,
                                           startValue=0.0001,
                                           increment=0.0001,
                                           upperValue=0.01):
```

```

        df, featureLabel, valueLabel = des.getModelFeatureImportance(self.getFinalModel())

        retDf = des.analyzeModelFeatureImportance(data=df,
                                                  valueLabel=valueLabel,
                                                  startValue=startValue,
                                                  increment=increment,
                                                  upperValue=upperValue,
                                                  returnAbove=returnAbove,
                                                  showSummary=True)
        return retDf

    def showBaseModelFeatureImportance(self,
                                       startValue=0.0001,
                                       increment=0.0001,
                                       upperValue=0.01):

        df, featureLabel, valueLabel = des.getModelFeatureImportance(self.getBaseModel())

        des.analyzeModelFeatureImportance(data=df,
                                          startValue=startValue,
                                          increment=increment,
                                          upperValue=upperValue,
```

```python
                                          showSummary=False)

        des.showAllModelFeatureImportance(data=df,
                                          featureLabel=featureLabel,
                                          valueLabel=valueLabel
                                          )

    def showFinalModelFeatureImportance(self,
                                        startValue=0.0001,
                                        increment=0.0001,
                                        upperValue=0.01):

        df, featureLabel, valueLabel = des.getModelFeatureImportance(self.getFinalModel())

        des.analyzeModelFeatureImportance(data=df,
                                          startValue=startValue,
                                          increment=increment,
                                          upperValue=upperValue,
                                          showSummary=False)

        des.showAllModelFeatureImportance(data=df,
                                          featureLabel=featureLabel,
                                          valueLabel=valueLabel
                                          )

    def showBaseModelReport(self,
                            axisLabels
                            ):
        self.showBaseModelStats()

        des.showReport(data=self.getBaseModelPrediction(),
                       colNameActual=self.baseModelPredictionColActual,
                       colNamePredict=self.baseModelPredictionColPredict,
                       axisLabels=axisLabels,
                       titleSuffix=self.experimentName)

        self.showBaseModelLearningCurve()
        self.showBaseModelROCAUC(axisLabels=axisLabels)

    def showBaseModelROCAUC(self, axisLabels, useStored=False):
        if useStored and self.isBaseModelROCAUCCalculated:
            print('Base model ROCAUC already calculated. Displaying stored results')
            tViz = self.__getBaseModelROCAUC()
            tViz.show()
        else:
            print('Base model ROCAUC not calculated. Starting now')
            viz = des.showROCAUC(dataTrain=self.dataPackage.getTrainData(),
                                 dataTest=self.dataPackage.getTestData(),
                                 classifier=self.getClassifier(),
                                 axisLabels=axisLabels,
                                 colNameActual=self.dataPackage.targetColumn,
                                 features=self.getBaseFeatures())
            self.__setBaseModelROCAUC(visualizer=viz)
            viz.show()

    def __setBaseModelROCAUC(self,
                             visualizer):
        self.isBaseModelROCAUCCalculated = True
        self.baseModelROCAUC = pickle.dumps(visualizer)

    def __getBaseModelROCAUC(self):
        return pickle.loads(self.baseModelROCAUC)

    def showFinalModelROCAUC(self, axisLabels, useStored=False):
        if useStored and self.isFinalModelROCAUCCalculated:
            print('Final model ROCAUC already calculated. Displaying stored results')
            tViz = self.__getBaseModelROCAUC()
            tViz.show()
        else:
            print('Final model ROCAUC not calculated. Starting now')
            viz = des.showROCAUC(dataTrain=self.dataPackage.getTrainData(),
                                 dataTest=self.dataPackage.getTestData(),
                                 classifier=self.getClassifier(),
                                 axisLabels=axisLabels,
                                 colNameActual=self.dataPackage.targetColumn,
                                 features=self.
```

```
getFinalFeatures())
            self.__setFinalModelROCAUC(visualizer=viz)
            viz.show()

    def __getFinalModelROCAUC(self):
        return pickle.loads(self.finalModelROCAUC)

    def __setFinalModelROCAUC(self,
                              visualizer):
        self.isFinalModelROCAUCCalculated = True
        self.finalModelROCAUC = pickle.dumps(visualizer)

    def showFinalModelReport(self,
                             axisLabels
                             ):
        self.showFinalModelStats()

        des.showReport(data=self.getFinalModelPrediction(),
                       colNameActual=self.finalModelPredictionColActual,
                       colNamePredict=self.finalModelPredictionColPredict,
                       axisLabels=axisLabels,
                       titleSuffix=self.experimentName)

        self.showFinalModelLearningCurve()
        self.showFinalModelROCAUC(axisLabels=axisLabels)

    def getBaseFeatures(self):
        return self.dataPackage.dataFeatures

    def getFinalFeatures(self):
        return self.finalFeatures

```

```python
    def createBaseModelLearningCurve(self,
                                     cv=None,
                                     n_jobs=None,
                                     train_sizes=None,
                                     verbose=4):
        # If it is already predicted just show it
        if self.isBaseModelLearningCurveCreated:
            print('Base model learning curve already calculated. Displaying results:')
            self.showBaseModelLearningCurve()
        else:
            df = self.dataPackage.getTrainData()
            train_sizes, train_scores, test_scores, fit_times = des.create_learning_curve(
                estimator=self.getClassifier(),
                X=df[self.dataPackage.dataFeatures],
                y=df[self.dataPackage.targetColumn],
                cv=cv,
                n_jobs=n_jobs,
                train_sizes=train_sizes,
                verbose=verbose)

            self.__setBaseModelLearningData(train_sizes=train_sizes,
                                            train_scores=train_scores,
                                            test_scores=test_scores,
                                            fit_times=fit_times)

    def createFinalModelLearningCurve(self,
                                      cv=None,
                                      n_jobs=None,
                                      train_sizes=None,
                                      verbose=4):
        # If it is already predicted just show it
        if self.isFinalModelLearningCurveCreated:
            print('Final model learning curve already calculated. Displaying results:')
            self.showFinalModelLearningCurve()
        else:

            df = self.dataPackage.getTrainData()
            train_sizes, train_scores, test_scores, fit_times = des.create_learning_curve(
                estimator=self.getClassifier(),
                X=df[self.finalFeatures],
                y=df[self.dataPackage.targetColumn],
                cv=cv,
                n_jobs=n_jobs,
                train_sizes=train_sizes,
                verbose=verbose)
            self.__setFinalModelLearningData(train_sizes=train_sizes,
                                             train_scores=train_scores,
                                             test_scores=test_scores,
                                             fit_times=fit_times)

    def __setBaseModelLearningData(self,
                                   train_sizes,
                                   train_scores,
                                   test_scores,
                                   fit_times):
        self.isBaseModelLearningCurveCreated = True
        self.baseModel_train_sizes = train_sizes
        self.baseModel_train_scores = train_scores
        self.baseModel_test_scores = test_scores
        self.baseModel_fit_times = fit_times

    def __setFinalModelLearningData(self,
                                    train_sizes,
                                    train_scores,
                                    test_scores,
                                    fit_times):
        self.isFinalModelLearningCurveCreated = True
        self.finalModel_train_sizes = train_sizes
        self.finalModel_train_scores = train_scores
        self.finalModel_test_scores = test_scores
        self.finalModel_fit_times = fit_times

    def showBaseModelLearningCurve(self,
                                   axes=None,
                                   ylim=(0.0, 1.01)
                                   ):
        if self.isBaseModelLearningCurveCreated:

            des.plot_learning_curve(train_sizes=self.baseModel_train_sizes,
                                    train_scores=self.baseModel_train_scores,
                                    test_scores=self.baseModel_test_scores,
                                    fit_times=self.baseModel_fit_times,
                                    title=self.experimentName,
                                    axes=axes,
                                    ylim=ylim
                                    )
        else:
            display('Base model Learning curve has not yet been calculated')

    def showFinalModelLearningCurve(self,
                                    axes=None,
                                    ylim=(0.0, 1.01)
                                    ):
        if self.isFinalModelLearningCurveCreated:
            des.plot_learning_curve(train_sizes=self.
```

```python
finalModel_train_sizes,
                                    train_scores=self.finalModel_train_scores,
                                    test_scores=self.finalModel_test_scores,
                                    fit_times=self.finalModel_fit_times,
                                    title=self.experimentName,
                                    axes=axes,
                                    ylim=ylim
                                    )
        else:
            display('Final model Learning curve has not yet been calculated')

    def __getFinalModelFeatures(self,
                                returnAbove=0.002,
                                includeUniqueAndTarget=False):
        # get a list of the features that have been deemed important
        # Get full list of features
        features = self.dataPackage.dataFeatures

        df, featureLabel, valueLabel = des.getModelFeatureImportance(self.getBaseModel())

        retDf = des.analyzeModelFeatureImportance(data=df,
                                                  valueLabel=valueLabel,
                                                  returnAbove=returnAbove,
                                                  showSummary=False,
                                                  showPlot=False)

        keepFeatures = retDf[featureLabel].to_list()

        # Initialize important features list
        features_important = []

        for x in keepFeatures:
            features_important.append(features[x])

        if includeUniqueAndTarget:
            # Feature list doesn't include target and unique
            features_important.append(self.dataPackage.uniqueColumn)
            features_important.append(self.dataPackage.targetColumn)

        return features_important

```