```python
import numpy as np
import pandas as pd
import decimal
import seaborn as sns
import matplotlib.pyplot as plt
import copy
from sklearn import metrics
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import cohen_kappa_score
from sklearn.model_selection import learning_curve
from sklearn.model_selection import StratifiedKFold
from tqdm.notebook import tqdm
from yellowbrick.classifier import ROCAUC


def createModel(data, uniqueColumn, targetColumn, classifier):
    tDf = data.copy()

    # Get Y value from dataframe
    Y = np.array(tDf[targetColumn])

    # Drop unneeded columns for model training
    tDf.drop([uniqueColumn, targetColumn], axis=1, inplace=True)

    # Get X Value from rest of dataframe
    X = tDf.to_numpy()

    # fit model on training data
    model = copy.deepcopy(classifier)
    model.fit(X, Y)

    return model
```

```python


def predictModel(model,
                 data,
                 uniqueColumn,
                 targetColumn,
                 colActual='y_test',
                 colPredict='y_pred'):
    tDf = data.copy()

    Y = np.array(tDf[targetColumn])

    # Drop unneeded columns for model testing
    tDf.drop([uniqueColumn, targetColumn], axis=1, inplace=True)

    # Get X Value from rest of dataframe
    X = tDf.to_numpy()

    y_pred = model.predict(X)

    # make a dataframe for the results
    tDf = pd.DataFrame(data=Y, columns=[colActual])
    tDf[colPredict] = y_pred.tolist()

    return tDf, colActual, colPredict


# Creates a dataframe with two columns:
# feature_idx: index of features
# importance: importance value of the relevent feature
def getModelFeatureImportance(model,
                              featureLabel='feature_idx',
                              valueLabel='importance'):
    # Create a dataframe with feature importances
    impDf = pd.DataFrame(data=model.feature_importances_, columns=[valueLabel])
    impDf.reset_index(inplace=True)
    impDf.rename(columns={'index': featureLabel}, inplace=True)

    return impDf, featureLabel, valueLabel


# plots importance of features in given model
def analyzeModelFeatureImportance(data,
                                  valueLabel='importance',
                                  startValue=0.0001,
                                  increment=0.0001,
                                  upperValue=0.01,
                                  returnAbove=0.002,
                                  showSummary=True,
                                  showPlot=True):
    tqdm.pandas()
    xAxisLabel = 'Feature Importance'
    recCountLabel = 'Number of Documents'
    dx = startValue

    # calc the number of decimals to round the value
    d = decimal.Decimal(str(increment))
    roundValue = d.as_tuple().exponent * -1

    # Create the list with initial value of 0
    xAxisVal = [startValue]

    while dx <= upperValue:
        # add to the list of xAxisValues
        xAxisVal.append(dx)

        dx += increment
        # round value included due to errors in FP addition
        dx = round(dx, roundValue)

    # turn list into dataframe
    tDf = pd.DataFrame(xAxisVal, columns=[xAxisLabel])

    # Add in column for number of features >= that value
    tDf[recCountLabel] = tDf.progress_apply(lambda x:
                                            len(data.loc[data[valueLabel] >= x[xAxisLabel]]),
                                            axis=1
                                            )

    # return a list of features to be used in an optimized model
    # it's by feature index
    tDf2 = data.loc[data[valueLabel] >= returnAbove].copy()
    tDf2.reset_index(drop=True, inplace=True)

    if showSummary:
        # Give some sort of summary
        indent = '---> '
        print('Feature Importance Summary:')
        print(f'{indent}Original feature count: {len(data)}')
        print(f'{indent}Returned feature count: {len(tDf2)}')
        print(f'{indent}Removed feature count: {len(data) - len(tDf2)}')
        print(f'{indent}Return items above (including): {returnAbove}')

    if showPlot:
        # Plot it after the summary
        tDf.plot(x=xAxisLabel,
                 y=recCountLabel,
                 ylabel='Cumulative count of documents',
                 title='Total Documents >= Importance
```

```
Level')

    return tDf2


def showAllModelFeatureImportance(data,
                                  featureLabel,
                                  valueLabel,
                                  xlim=None):
    #if len(data) > 100:
    #    recLimit = 25
    #else:
    #    recLimit = max(len(data),25)

    recLimit = 25

    if xlim is None:
        xlim = .03

    newFeatLabel = featureLabel + '_s'
    tDf = data.sort_values(by=valueLabel, ascending=False).head(recLimit).copy()
    tDf[newFeatLabel] = tDf.apply(lambda x:
                                  'feature_' + str(x[featureLabel]),
                                  axis=1
                                  )

    sns.set_theme(style="whitegrid")

    # Initialize the matplotlib figure
    f, ax = plt.subplots(figsize=(6, 10))

    # Plot the todtal crashes
    # sns.set_color_codes("pastel")
    sns.barplot(x=valueLabel,
                y=newFeatLabel,
                data=tDf,
                label="Total",
                palette="crest").set(title=f'Model Feature Importance (top {recLimit})')

    plt.tick_params(axis='y', labelsize=10)

    # ax.legend(ncol=1, loc="lower right", frameon=True)
    ax.set(xlim=(0, xlim),
           ylabel="",
           xlabel="Feature Importance")
    # sns.despine(left=True, bottom=True)
    sns.despine()


def showConfusionMatrix(data,
                        colNameActual,
                        colNamePredict,
                        axis_labels,
                        titleSuffix,
                        cmap='mako',
                        plotsize=2):
    plt.clf()
    cm = confusion_matrix(np.array(pd.to_numeric(data[colNameActual])).reshape(-1, 1),
                          np.array(pd.to_numeric(data[colNamePredict])).reshape(-1, 1)
                          )

    sns.heatmap(cm,
                annot=True,
                fmt='d',
                cmap=cmap,
                xticklabels=axis_labels,
                yticklabels=axis_labels
                )

    if plotsize == 5:
```

```python
        sns.set(rc={'figure.figsize': (20, 8)})
    elif plotsize == 4:
        sns.set(rc={'figure.figsize': (15, 8)})
    elif plotsize == 3:
        sns.set(rc={'figure.figsize': (10, 8)})
    elif plotsize == 2:
        sns.set(rc={'figure.figsize': (8, 8)})
    elif plotsize == 1:
        sns.set(rc={'figure.figsize': (4, 8)})
    else:  # Should be size 1
        # should only be one but catch it and default to size 1
        sns.set(rc={'figure.figsize': (4, 4)})

    # title with fontsize 20
    plt.title(f'Confusion Matrix: {titleSuffix}', fontsize=20)
    # x-axis label with fontsize 15
    plt.xlabel('Predicted', fontsize=15)
    # y-axis label with fontsize 15
    plt.ylabel('Actual', fontsize=15)
    plt.show()
    plt.clf()


def showReport(data, colNameActual, colNamePredict, axisLabels, titleSuffix):
    results = metrics.classification_report(pd.to_numeric(data[colNameActual]).to_list(),
                                            data[colNamePredict].to_list(),
                                            zero_division=0)
    print(results)

    showConfusionMatrix(data=data,
                        colNameActual=colNameActual,
                        colNamePredict=colNamePredict,
                        axis_labels=axisLabels,
                        titleSuffix=titleSuffix
                        )

def showROCAUC(dataTrain,
               dataTest,
               classifier,
               axisLabels,
               colNameActual,
               features):

    model = classifier
    visualizer = ROCAUC(model, classes=axisLabels)

    # Fit the training data to the visualizer
    visualizer.fit(dataTrain[features],
                   dataTrain[colNameActual]
                   )
    # Evaluate model
    visualizer.score(dataTest[features],
                     dataTest[colNameActual]
                     )

    visualizer.show()

    return visualizer

def create_learning_curve(estimator,
                          X,
                          y,
                          cv=None,
                          n_jobs=None,
                          train_sizes=None,
                          verbose=4):
    if train_sizes is None:
        train_sizes = [0.1, 0.2, 0.5, 1.0]

    if cv is None:
        cv = StratifiedKFold(n_splits=5, random_state=0, shuffle=True)

    train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
        estimator,
        X,
        y,
        cv=cv,
        n_jobs=n_jobs,
        train_sizes=train_sizes,
        return_times=True,
        verbose=verbose
    )
    return train_sizes, train_scores, test_scores, fit_times


def plot_learning_curve(train_sizes,
                        train_scores,
                        test_scores,
                        fit_times,
                        title,
                        axes=None,
                        ylim=None
                        ):
    """
    Generate 3 plots: the test and training learning curve, the training
    samples vs fit times curve, the fit times vs score curve.
    """
    plt.clf()

    if axes is None:
        _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title(title)
    if ylim is not None:
        axes[0].set_ylim(*ylim)
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

    # Plot learning curve
    axes[0].grid()
    axes[0].fill_between(
        train_sizes,
        train_scores_mean - train_scores_std,
        train_scores_mean + train_scores_std,
        alpha=0.1,
        color="r",
    )
    axes[0].fill_between(
        train_sizes,
        test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std,
        alpha=0.1,
        color="g",
    )
    axes[0].plot(
        train_sizes, train_scores_mean, "o-", color="r", label="Training score"
    )
    axes[0].plot(
        train_sizes, test_scores_mean, "o-", color="g", label="Cross-validation score"
    )
    axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, "o-")
    axes[1].fill_between(
        train_sizes,
        fit_times_mean - fit_times_std,
        fit_times_mean + fit_times_std,
        alpha=0.1,
    )
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("fit_times")
    axes[1].set_title("Scalability of the model")

    # Plot fit_time vs score
    fit_time_argsort = fit_times_mean.argsort()
    fit_time_sorted = fit_times_mean[fit_time_argsort]
    test_scores_mean_sorted = test_scores_mean[fit_time_argsort]
    test_scores_std_sorted = test_scores_std[fit_time_argsort]
    axes[2].grid()
    axes[2].plot(fit_time_sorted, test_scores_mean_sorted, "o-")
    axes[2].fill_between(
        fit_time_sorted,
        test_scores_mean_sorted - test_scores_std_sorted,
        test_scores_mean_sorted + test_scores_std_sorted,
        alpha=0.1,
    )
    axes[2].set_xlabel("fit_times")
    axes[2].set_ylabel("Score")
    axes[2].set_title("Performance of the model")

    plt.show()
    plt.clf()

```

```

def getModelAccuracy(data, colActual, colPredict):
    accuracy = accuracy_score(data[colActual],
                              data[colPredict])

    return accuracy


def getModelPrecision(data, colActual, colPredict,
average='weighted'):
    precision = precision_score(data[colActual],
                                data[colPredict],
                                average=average)

    return precision


def getModelRecall(data, colActual, colPredict,
average):
    recall = recall_score(data[colActual],
                          data[colPredict],
                          average=average)

    return recall


def getModelF1(data, colActual, colPredict, average):
    f1 = f1_score(data[colActual],
                  data[colPredict],
                  average=average)

    return f1


def getModelCohenKappa(data, colActual, colPredict):
    ck = cohen_kappa_score(data[colActual],
                           data[colPredict])

    return ck

```