An Empirical Comparison of Supervised Learning Algorithms for Classification

Dillon Ford, PID: A16092047

***Abstract*—This paper examines multiple supervised machine learning algorithms used to solve classification problems. Here we perform an empirical analysis and comparison between the supervised learning methods: logistic regression, k-nearest neighbors, random forests, and decision trees. Performance is measured across multiple trials and datasets for each classifier.**

I. Introduction

For a classification problem, finding the appropriate classifier with optimal parameters is crucial for success. Many of the ideas for this paper stem from prior work conducted to introduce methodologies and practices in classification [1] that optimized and empirically analyzed the performance of classifiers over many datasets. This paper will examine four different classifiers' performance in three trials across three datasets to solve supervised machine learning algorithms' problems. The Mushroom, Heart Disease, and Drug Consumption datasets are obtainable from the UCI Machine Learning Repository [2].

A note on datasets: The instructor provided approval for using datasets in this paper.

The Mushroom dataset (MU) contains records drawn from The Audubon Society Field Guide to North American Mushrooms [3]. The goal is to distinguish edible mushrooms from poisonous mushrooms based on 22 features describing the mushroom characteristics such as odor, ring type, and gill size and color. MU classification problem was attractive to me because I lived in the Pacific Northwest for some time and became fascinated with hunting and identifying mushrooms in the Redwood Forest.

The Heart Disease dataset (HD) contains records from medical centers in Budapest, Switzerland, and California [4]. The classification operation is to separate patients with heart disease from patients without heart disease utilizing 14 different attributes including, chest pain type, resting blood pressure, and age, to name a few. Being able to classify whether or not a patient has heart disease is crucial as heart disease is the number one leading cause of death in the United States [5] which is why I elected to run this classification problem.

The Drug Consumption dataset (DRG) contains records from institutes in the UK, including Rampton Hospital, University of Nottingham, and University of Leicester [6]. Respondents were questioned about their use of 18 legal and illegal drugs and asked about how frequently they used the drug. For this project, a transformation of the dataset into a binary classification problem took place to examine whether or not the participant is a "user" or "non-user" of a particular drug based on 32 attributes such as the big five personality traits. The drug examined in this project is marijuana since it is the most balanced in terms of use and non-use in this dataset. Favorite topics of mine to study are pharmacology and the psychology of drug use. I was interested in seeing whether or not personality and behavior could determine whether or not a person uses a particular drug.

II. Methods

Four learning algorithms are tested across the datasets consisting of three trials in each. Learning algorithms include Logistic Regression, K-Nearest Neighbors, Random Forests, and Decision Trees. Modeling of the algorithm parameters come from the Caruna and Niculescu-Mizil paper [1] as described below. Implementation of Scikit-learn's GridSearchCV [7] is used to find the optimal parameters associated with each algorithm across each trial. The performance of each learning algorithm is evaluated using the performance metrics: accuracy and precision. Scores between all metrics are averaged across the three trials for each dataset and averaged between the three datasets.

The parameter spaces searched for each of the classifiers in each of the datasets are shown below.

Logistic Regression (LOGREG): in this learning algorithm, regularized and unregularized models are implemented. The regularization parameter C varies by factors of ten from 10^{-8} to 10^{4}.

K-Nearest Neighbors (KNN): in this learning algorithm, KNN with Euclidean distance is implemented and weighted uniformly and by distance. The number of neighbors is 20, varying by 15, ranging from 1 to 300 for MU and DRG. Neighbor values adjusted to 20 varying by 5, ranging from 1 to 60 for HD due to it being a significantly smaller dataset.

Random Forests (RF): in this learning algorithm, the forests have 1024 trees. The size of the feature set considered at each split is 1,2,4,6,8,12,16, or 20.

Decision Trees (DT): in this learning algorithm, the decision tree classifier CART with the gini criterion is used with a max depth of each ranging from 1 to 10. The best splitting method is used with 2 samples to split.

III. Experiment

For each dataset and learning algorithm, 3 trials are performed. For each trial in the datasets, randomly select 80 percent of the training cases and the remaining 20 percent of cases for testing. A Pipeline GridSearchCV for all learning algorithms is performed with 5-fold cross-validation on the training cases to return the best parameters for the mean over 5-folds. Tuning of GridSearchCV for scoring metrics occurred to refit for accuracy and precision. In each case, precision and accuracy, the model is trained a final time on all the training cases, selecting optimal parameters, and measuring performance on the test cases.[1]

For each trial and learning algorithm, visualization of training performance is presented through a learning curve, plotting training score against cross-validation score across training examples, a plot for the scalability of the model, showing fit times across training examples, and a plot of model performance, showing score across fit times. Please see the Appendix for an example of such graphics.

IV. Results

Table 1 . Algorithm Mean Test Set Performance By Dataset

		KNN	RF	LR	DT
HD	**Accuracy**	0.809	0.831	0.847	0.781
	Precision	0.789	0.855	0.839	0.794
DRG	**Accuracy**	0.801	0.791	0.811	0.774
	Precision	0.857	0.845	0.850	0.851
MU	**Accuracy**	1.000	1.000	0.966	1.000
	Precision	1.000	1.000	0.964	1.000

In the HD dataset, LOGREG performed best in accuracy, and RF showed the highest precision scores. Similarly, in DRG, LOGREG also showed the highest accuracy results, but KNN scored the highest in precision. In contrast, MU, LOGREG did not perform as well as other learning algorithms. Table 1 summarizes the results of each dataset and algorithm accuracy and precision performance. A paired t-test was used between each algorithm across the three datasets to find the p-value, and a calculation of Cohen's d-value to measure effect size; these values are in Appendix 1 along with dataset raw algorithm test scores for the trial. With $p = 0.05$, significant results are shown in DRG, where a comparison between KNN and DT for accuracy resulted in $p = 0.035$ and $d = 2.610$ as well as LOGREG and DT, with $p = 0.007$ and $d = 4.583$. In MUSH significant results are shown between LR and all other classifiers, with $p = 0.01$ and $d = 8.057$ for accuracy, and $p = 0.026$ and $d = 4.978$ for precision.

[1]A .py file containing functions used to perform these calculations and a notebook outlining execution and analysis are in the Appendix; for complete documentation of code and notebooks, please refer to my GitHub, https://github.com/dillon4d/COGS118A_FinalProject

Table 2 . Algorithm Mean Test Set Performance

	KNN	RF	LR	DT
Accuracy	0.870	0.874	0.875	0.852
Precision	0.882	0.900	0.885	0.881

An analysis of the overall performance of each classification algorithm ensued. In terms of accuracy, LOGREG and RF performed almost equally, followed by KNN and DT. For precision, RF outperformed all other learning algorithms. Table 2 summarizes the mean test set performance for each algorithm. As in Appendix 1, respective p-values and d-values are in Appendix 2. With $p = 0.05$, no significant results were shown between classifiers when averaging test performances across the three datasets, and there was no significance in effect size.

Table 3 . Algorithm Mean Optimal Train Set Performance

		KNN	RF	LR	DT
HD	**Accuracy**	0.814	0.821	0.817	0.749
	Precision	0.856	1.000	0.815	0.864
DRG	**Accuracy**	0.802	0.807	0.813	0.723
	Precision	0.903	1.000	0.857	0.904
MU	**Accuracy**	0.999	1.000	0.965	0.998
	Precision	1.000	1.000	0.964	1.000

In comparison to algorithm mean test set performance, as in Table 1, scores in Table 3 reflect the mean train set performance of algorithms adjusted with optimal hyperparameters. In HD, algorithm accuracy performance was similar to test set performance; KNN accuracy slightly increased while the other algorithms slightly decreased. Precision scores improved significantly for HD classifiers when adjusted, notably RF, while LR saw a slight decrease. In DRG, accuracy performance increased somewhat in all classifiers with adjusted parameters except DT, and precision scores improved in all classifiers. In MU, performance with adjusted parameters in the train set remained nearly the same for all classifiers as in their test set performance. Raw scores for dataset algorithm trainset with optimal parameters are in Appendix 3.

V. Conclusion

Overall, RF showed the most precise results among the four classifiers tested across datasets and had near equal accuracy to that of LR, which showed the highest mean test accuracy performance. As shown in Appendix Table 2, the t-test on average performances between algorithms shows no significant results nor significance in effect size. In retrospect, there is room for improvement; the testing of more classifiers, trials, and metrics would provide insight into the results achieved here. Further assessment of the datasets utilizing a bootstrapping analysis would also prove to be beneficial. Given more time, a more in-depth analysis of more drugs, not just marijuana, in DRG could provide greater insight into the personality and the attractiveness of drugs.

VI. Bonus Points

This project deserves bonus points for performing tests on an extra learning algorithm and evaluating the algorithms on an accuracy metric and a precision metric. Furthermore, during each trial, for each algorithm, learning curves are plotted to show performance and other visualizations for each trial algorithm, such as a confusion matrix and ROC curve. Each notebook contains an extensive exploratory data analysis, including comprehensive data visualizations. In HD and MU, perform a preemptive random forest classifier to retrieve the dataset's top features. The top 5 explored extensively before model training.

References

[1] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 161–168. [Online]. Available: https://doi.org/10.1145/1143844.1143865

[2] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[3] G. H. Lincoff, *The Audubon Society Field Guide to North American Mushrooms*. New York: Alfred A. Knopf, 1981.

[4] R. Detrano, A. Janosi, W. Steinbrunn, M. Pfisterer, J.-J. Schmid, S. Sandhu, K. H. Guppy, S. Lee, and V. Froelicher, "International application of a new probability algorithm for the diagnosis of coronary artery disease," *The American Journal of Cardiology*, vol. 64, no. 5, pp. 304 – 310, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0002914989905249

[5] N. S. Shah, R. Molsberry, J. S. Rana, S. Sidney, S. Capewell, M. O'Flaherty, M. Carnethon, D. M. Lloyd-Jones, and S. S. Khan, "Heterogeneous trends in burden of heart disease mortality by subtypes in the united states, 1999-2018: observational analysis of vital statistics," *BMJ*, vol. 370, 2020. [Online]. Available: https://www.bmj.com/content/370/bmj.m2688

[6] E. Fehrman, A. K. Muhammad, E. M. Mirkes, V. Egan, and A. N. Gorban, "The five factor model of personality and evaluation of drug consumption risk," in *Data Science*, F. Palumbo, A. Montanari, and M. Vichi, Eds. Cham: Springer International Publishing, 2017, pp. 231–242.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Appendix

Below the appendix tables is a .py file that outlines much of the learning algorithms' calculations (varies slightly by metric and dataset). Also attached is a trial from one of the notebooks. This process is repeated across all three trials in every dataset. The full documentation of notebooks for the datasets and their .py files are found on my GitHub.

https://github.com/dillon4d/COGS118A_FinalProject

Appendix 1 . p-values and d-values for algorithm comparisons by dataset

Dataset	Metric	KNN & RF	KNN & LR	KNN & DT	RF & DT	RF & LR	LR & DT
HD	Accuracy	p = 0.746	p = 0.463	p = 0.607	p = 0.502	p = 0.814	p = 0.395
		d = 0.289	d = 0.664	d = 0.456	d = 0.608	d = 0.099	d = 0.780
	Precision	p = 0.363	p = 0.297	p = 0.922	p = 0.438	p = 0.827	p = 0.459
		d = 0.907	d = 1.019	d = 0.088	d = 0.708	d = 0.193	d = 0.671
DRG	Accuracy	p = 0.380	p = .291	p = 0.035	p = 0.127	p = 0.097	p = 0.007
		d = 0.807	d = 1.030	d = 2.61	d = 1.601	d = 1.937	d = 4.583
	Precision	p = 0.609	p = 0.679	p = 0.757	p = 0.821	p = .809	p = 0.982
		d = 0.456	d = 0.368	d = 0.272	d = 0.198	d = 0.216	d = 0.020
MU	Accuracy	p = nan	p = 0.010	p = nan	p = nan	p = 0.010	p = 0.010
		d = nan	d = 8.057	d = nan	d = nan	d = 8.057	d = 8.057
	Precision	p = nan	p = 0.026	p = nan	p = nan	p = 0.026	p = 0.026
		d = nan	d = 4.978	d = nan	d = nan	d = 4.978	d = 4.978

Appendix 1.1. Raw Test Set Scores By Dataset

		KNN	RF	LR	DT
HD	Accuracy	Trial 1: 0.787	Trial 1: 0.803	Trial 1: 0.803	Trial 1: 0.771
		Trial 2: 0.771	Trial 2: 0.754	Trial 2: 0.820	Trial 2: 0.721
		Trial 3: 0.869	Trial 3: 0.934	Trial 3: 0.918	Trial 3: 0.853
	Precision	Trial 1: 0.750	Trial 1: 0.815	Trial 1: 0.793	Trial 1: 0.759
		Trial 2: 0.800	Trial 2: 0.784	Trial 2: 0.816	Trial 2: 0.744
		Trial 3: 0.816	Trial 3: 0.967	Trial 3: 0.909	Trial 3: 0.879
DRG	Accuracy	Trial 1: 0.796	Trial 1: 0.801	Trial 1: 0.804	Trial 1: 0.775
		Trial 2: 0.814	Trial 2: 0.796	Trial 2: 0.817	Trial 2: 0.783
		Trial 3: 0.793	Trial 3: 0.777	Trial 3: 0.812	Trial 3: 0.764
	Precision	Trial 1: 0.875	Trial 1: 0.878	Trial 1: 0.859	Trial 1: 0.868
		Trial 2: 0.834	Trial 2: 0.827	Trial 2: 0.835	Trial 2: 0.823
		Trial 3: 0.861	Trial 3: 0.832	Trial 3: 0.857	Trial 3: 0.861
MU	Accuracy	Trial 1: 1.000	Trial 1: 1.000	Trial 1: 0.971	Trial 1: 1.000
		Trial 2: 1.000	Trial 2: 1.000	Trial 2: 0.967	Trial 2: 1.000
		Trial 3: 1.000	Trial 3: 1.000	Trial 3: 0.959	Trial 3: 1.000
	Precision	Trial 1: 1.000	Trial 1: 1.000	Trial 1: 0.972	Trial 1: 1.000
		Trial 2: 1.000	Trial 2: 1.000	Trial 2: 0.968	Trial 2: 1.000
		Trial 3: 1.000	Trial 3: 1.000	Trial 3: 0.953	Trial 3: 1.000

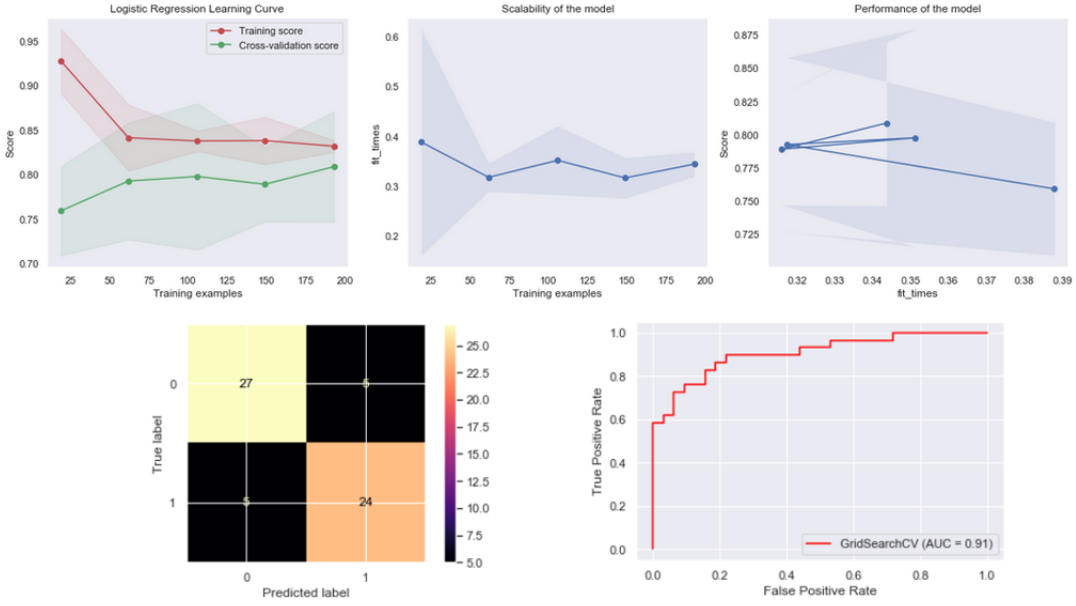
Appendix Table 2. p-values and d-values for algorithm comparisons

	KNN & RF	KNN & LR	KNN & DT	RF & DT	RF & LR	LR & DT
Accuracy	p = 0.935	p = 0.914	p = 0.727	p = 0.678	p = 0.989	p = 0.630
	d = 0.038	d = 0.052	d = 0.167	d = 0.200	d = 0.006	d = 0.232
Precision	p = 0.680	p = 0.942	p = 0.994	p = 0.682	p = 0.687	p = 0.937
	d = 0.198	d = 0.035	d = 0.004	d = 0.197	d = 0.194	d = 0.038

Appendix 3. Raw Optimal Train Set Scores By Dataset

		KNN	RF	LR	DT
HD	**Accuracy**	Trial 1: 0.827 Trial 2: 0.826 Trial 3: 0.789	Trial 1: 0.818 Trial 2: 0.831 Trial 3: 0.814	Trial 1: .831 Trial 2: 0.806 Trial 3: 0.814	Trial 1: 0.736 Trial 2: 0.814 Trial 3: 0.699
	Precision	Trial 1: 0.825 Trial 2: 0.872 Trial 3: 0.871	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000	Trial 1: 0.813 Trial 2: 0.840 Trial 3: 0.791	Trial 1: 0.744 Trial 2: 0.970 Trial 3: 0.879
DRG	**Accuracy**	Trial 1: 0.801 Trial 2: 0.801 Trial 3: 0.804	Trial 1: 0.810 Trial 2: 0.803 Trial 3: 0.808	Trial 1: 0.816 Trial 2: 0.813 Trial 3: 0.812	Trial 1: 0.787 Trial 2: 0.694 Trial 3: 0.688
	Precision	Trial 1: 0.858 Trial 2: 0.852 Trial 3: 1.000	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000	Trial 1: 0.855 Trial 2: 0.860 Trial 3: 0.855	Trial 1: 0.903 Trial 2: 0.909 Trial 3: 0.898
MU	**Accuracy**	Trial 1: 0.999 Trial 2: 0.999 Trial 3: 0.999	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000	Trial 1: 0.962 Trial 2: 0.968 Trial 3: 0.964	Trial 1: 0.998 Trial 2: 1.000 Trial 3: 0.996
	Precision	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000	Trial 1: 0.961 Trial 2: 0.968 Trial 3: 0.963	Trial 1: 1.000 Trial 2: 1.000 Trial 3: 1.000

Example of visualization for an accuracy trial of LR in HD



```
import numpy as np
from sklearn.metrics import classification_report, plot_confusion_matrix, plot_roc_curve
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, roc_auc_sc
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler, MinMaxScaler

#KNN, RandomForest, SVM

# Initializing Classifiers

clf1 = KNeighborsClassifier()
clf2 = RandomForestClassifier(n_estimators = 1024)
clf3 = LogisticRegression()
clf4 = tree.DecisionTreeClassifier()

# Building the pipelines

pipe1 = Pipeline([('std', StandardScaler()),
                  ('classifier', clf1)])

pipe2 = Pipeline([('std', StandardScaler()),
                 ('classifier', clf2)])

pipe3 = Pipeline([('std', StandardScaler()),
                  ('classifier', clf3)])

pipe4 = Pipeline([('std', StandardScaler()),
                  ('classifier', clf4)])

# Declaring some parameter values
C_list = np.power(10., np.arange(-8, 4))
D_list = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
F_list = [1, 2, 4, 6, 8, 12, 16, 20]
G_list = [0.001,0.005,0.01,0.05,0.1,0.5,1,2]
K_list = [n*15 for n in range(1,21)]
penalty_list = ['l1','l2']
weight_list = ['uniform','distance']

```

```
# list of the param lists
all_param_lists = [C_list, D_list, F_list, G_list, G_list, K_list, penalty_list, weight_list]
knn_param_list = [weight_list, K_list]
lr_param_list =[penalty_list, C_list]

# Setting up the parameter grids

param_grid1 = [{'classifier__weights': ['uniform', 'distance'],
                'classifier__n_neighbors': K_list}]

param_grid2= [{'classifier__max_features': F_list}]

param_grid3 = [{'classifier__C': C_list,
                'classifier__penalty': ['l1','l2']}]

param_grid4 = [{'classifier__max_depth':D_list}]

# Setting up multiple GridSearchCV objects, 1 for each algorithm

scoring_metrics = ['accuracy','precision', 'recall', 'f1']
gridcvs = {}

for pgrid, est, name in zip((param_grid1, param_grid2, param_grid3, param_grid4),
                            (pipe1, pipe2, pipe3, pipe4),
                            ('KNN','RandomForest','Logistic', 'DecisionTree')):
    gcv = GridSearchCV(estimator=est,
                       param_grid=pgrid,
                       scoring=scoring_metrics,
                       n_jobs=3,
                       cv=5, # 5-fold inner
                       verbose=0,
                       return_train_score=True,
                       refit='accuracy')
    gridcvs[name] = gcv

cv_scores = {name: [] for name, gs_est in gridcvs.items()}
skfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)

def train_algo(gridcvs, X_train, y_train, folds):
    """
    Train the algorithm on the classifiers and parameters defined in the gridsearch and returns
    parameters

    gridcvs = the setup gridsearchCV to train the algorithms on
    X_train, y_train = data split (random state of data varies by trial)
    folds = number of k folds to perform on the data
    """

```

```
    cv_scores = {name: [] for name, gs_est in gridcvs.items()}
    skfold = StratifiedKFold(n_splits=folds, shuffle=True, random_state=1)
    # The outer loop for algorithm selection
    c = 1
    for outer_train_idx, outer_valid_idx in skfold.split(X_train,y_train):
        for name, gs_est in sorted(gridcvs.items()):
            print('outer fold %d/5 | tuning %-8s' % (c, name), end='')
            # The inner loop for hyperparameter tuning
            gs_est.fit(X_train[outer_train_idx], y_train[outer_train_idx])
            y_pred = gs_est.predict(X_train[outer_valid_idx])
            acc = accuracy_score(y_true=y_train[outer_valid_idx], y_pred=y_pred)
            print(' | inner ACC %.2f%% | outer ACC %.2f%%' %
                  (gs_est.best_score_ * 100, acc * 100))
            cv_scores[name].append(acc)
        c += 1

    for name in cv_scores:
        print('%-8s | outer CV acc. %.2f%% +\- %.3f' % (
              name, 100 * np.mean(cv_scores[name]), 100 * np.std(cv_scores[name])))
    print()
    for name in cv_scores:
        print('{} best parameters'.format(name), gridcvs[name].best_params_)

def t1_algo(algo):

    if algo =='KNN':
        t1_algo_knn = gridcvs[algo]
        return t1_algo_knn

    elif algo == 'RandomForest':
        t1_algo_rf = gridcvs[algo]
        return t1_algo_rf

    elif algo == 'Logistic':
        t1_algo_lr = gridcvs[algo]
        return t1_algo_lr

    elif algo == 'DecisionTree':
        t1_algo_dt = gridcvs[algo]
        return t1_algo_dt

def t2_algo(algo):

    if algo =='KNN':
        t2_algo_knn = gridcvs[algo]
        return t2_algo_knn

    elif algo == 'RandomForest':
```

```
        t2_algo_rf = gridcvs[algo]
        return t2_algo_rf

    elif algo == 'Logistic':
        t2_algo_lr = gridcvs[algo]
        return t2_algo_lr

    elif algo == 'DecisionTree':
        t2_algo_dt = gridcvs[algo]
        return t2_algo_dt

def t3_algo(algo):

    if algo =='KNN':
        t3_algo_knn = gridcvs[algo]
        return t3_algo_knn

    elif algo == 'RandomForest':
        t3_algo_rf = gridcvs[algo]
        return t3_algo_rf

    elif algo == 'Logistic':
        t3_algo_lr = gridcvs[algo]
        return t3_algo_lr

    elif algo == 'DecisionTree':
        t3_algo_dt = gridcvs[algo]
        return t3_algo_dt

def fit_algo(trial_algo, algo, X_train, y_train, X_test, y_test, accu_dict, trial_accu, train_d
    """
    trial_algo = variable name associated with trial number and algorithm to store metrics, e.g
    algo = string of algo to get results of e.g. "SVM"
    X_train, X_test, y_train, y_test = data split (random state of data varies by trial)
    accu_dict = name of dictionary to store average accuracy across 5 folds
    trial_accu = label for value stored in accu_dict
    train_dict, test_dict = name of dictionaries to store train and test results
    """

    # Fitting model to the whole training set

    #using the "best" algorithm
    #trial_algo = gridcvs[algo]

    # fitting the algorithm to training data
    trial_algo.fit(X_train, y_train)
    #gathering train/test accuracy scores
    train_acc = accuracy_score(y_true=y_train, y_pred=trial_algo.predict(X_train))
```

```
    test_acc = accuracy_score(y_true=y_test, y_pred=trial_algo.predict(X_test))

    train_score_list = [train_acc]

    test_score_list = [test_acc]

    # print out results
    print('Accuracy %.2f%% (average over CV test folds)' % (100 * trial_algo.best_score_))
    print('Best Parameters: %s' % gridcvs[algo].best_params_)
    print('Training Accuracy: %.2f%%' % (100 * train_acc))
    print('Test Accuracy: %.2f%%' % (100 * test_acc))


    # store accuracy result average over CV test folds
    accu_dict[trial_accu] = trial_algo.best_score_

    # store results in dictionary using the  train_test_results function
    train_test_results(train_dict, algo, train_metric_list, train_score_list)
    train_test_results(test_dict, algo, test_metric_list, test_score_list)

#optimal parameters knn
def optimized(weight, K, F, C, D, penalty, X_train, y_train, X_test, y_test,train_dict,test_dic
    """
    weight = optimal weight parameter for KNN
    K = optimal number of n_neighbors for KNN
    F = optimal max_features for Random Forest
    C = optimal C value for Logistic Regression
    D = optimal max_depth of Decision Tree
    penalty = optimal 'l1' or 'l2' for Logistic Regression
    """

    clf_knn = KNeighborsClassifier(weights=weight, n_neighbors=K)
    clf_knn.fit(X_train,y_train)
    knn_accuracy = cross_val_score(clf_knn, X_train,y_train)
    pred_knn = clf_knn.predict(X_test)
    print('KNN Train Accuracy',np.mean(knn_accuracy))
    print('KNN Test Accuracy',clf_knn.score(X_test, y_test))
    print(classification_report(y_test, pred_knn))
    train_dict['KNN Train Accuracy'] = np.mean(knn_accuracy)
    test_dict['KNN Test Accuracy'] = clf_knn.score(X_test, y_test)

    clf_rf = RandomForestClassifier(n_estimators = 1024, max_features=F)
    clf_rf.fit(X_train,y_train)
    rf_accuracy = cross_val_score(clf_rf, X_train,y_train)
    pred_rf = clf_rf.predict(X_test)
    print('Random Forest Train Accuracy',np.mean(rf_accuracy))
    print('Random Forest Test Accuracy',clf_rf.score(X_test, y_test))
    print(classification_report(y_test, pred_rf))
```

```
    train_dict['Random Forest Train Accuracy'] = np.mean(rf_accuracy)
    test_dict['Random Forest Test Accuracy'] = clf_rf.score(X_test, y_test)

    clf_lr = LogisticRegression(penalty=penalty, C = C)
    clf_lr.fit(X_train,y_train)
    lr_accuracy = cross_val_score(clf_lr, X_train,y_train)
    pred_lr = clf_lr.predict(X_test)
    print('Logistic Train Accuracy',np.mean(lr_accuracy))
    print('Logistic Test Accuracy',clf_lr.score(X_test, y_test))
    print(classification_report(y_test, pred_lr))
    train_dict['Logistic Train Accuracy'] = np.mean(lr_accuracy)
    test_dict['Logistic Test Accuracy'] = clf_lr.score(X_test, y_test)

    clf_dt = tree.DecisionTreeClassifier(max_depth = D)
    clf_dt.fit(X_train,y_train)
    dt_accuracy = cross_val_score(clf_dt, X_train,y_train)
    pred_dt = clf_dt.predict(X_test)
    print('Decision Tree Train Accuracy',np.mean(dt_accuracy))
    print('Decision Tree Test Accuracy',clf_dt.score(X_test, y_test))
    print(classification_report(y_test, pred_dt))
    train_dict['Decision Tree Train Accuracy'] = np.mean(dt_accuracy)
    test_dict['Decision Tree Test Accuracy'] = clf_dt.score(X_test, y_test)

train_metric_list = ['Training Accuracy','Training Precision','Training Recall','Training F1','
test_metric_list = ['Test Accuracy','Test Precision','Test Recall','Test F1','Test ROC AUC']
#train_score_list = [train_acc, train_prec, train_recall, train_f1, train_roc]
#test_score_list = [test_acc, train_prec, test_prec, test_recall, test_f1, test_roc]

def train_test_results(dict_name,algo, metric_list, score_list):
    """
    dict_name = name of dictionary to put results
    algo = defined grid string of algo name to be used e.g. 'KNN','RandomForest', 'SVM', 'Logis
    metric_list = train/test metric list
    score_list = train/score lists defined in model fitting
    """
    for score, metric in zip(score_list, metric_list):
        dict_name[algo+' '+metric] = score

#train_string_list = ['mean_train_accuracy','mean_train_precision','mean_train_recall','mean_tr
#                     'mean_train_roc_auc',]

#test_string_list = ['mean_test_accuracy','mean_test_precision','mean_test_recall','mean_test_f
#                    'mean_test_roc_auc']

#train_metric_list = ['Training Accuracy','Training Precision','Training Recall','Training F1',
#test_metric_list = ['Test Accuracy','Test Precision','Test Recall','Test F1','Test ROC AUC']

def optimal_results(trial_algo, param_list, algo, param, dict_name, string_list, metric_list):
```

```
    """
    trial_algo = attained algorithm results for trial, e.g. t1_algo_svm => gridcvs[SVM]
    param_list = input optimal parameter's list for algo e.g C_list, K_list, D_list ect..
    algo = defined grid string of algo name to be used e.g. 'KNN','RandomForest', 'SVM', 'Logis
    param = corresponding parameter name to param_list e.g. C_list => 'classifier__C'
    dict_name = name of defined dictionary to store results in
    string_list = train_string_list for training results and test_string_list for testing resul
    metric_list = train_metric list for training and test_metric_list for testing
    """

    for i,j in enumerate(param_list):
        if j == gridcvs[algo].best_params_[param]:
            for item, metric in zip(string_list, metric_list):
                dict_name[algo+' '+metric] = trial_algo.cv_results_[item][i]

# Example Function Input:
#optimal_results(t1_algo_svm, C_list,'SVM', 'classifier__C',t1_optimal_train_dict, train_string

# Example Function Input:
#train_test_results(t1_train_dict, 'SVM', train_metric_list, train_score_list)
#train_test_results(t1_test_dict, 'SVM', test_metric_list, test_score_list)

# function to draw heat map of hyperparameters against performance
def draw_heatmap(acc, acc_desc, C_list, character):

    plt.figure(figsize = (2,4))
    ax = sns.heatmap(acc, annot=True, fmt='.3f', yticklabels=C_list, xticklabels=[])
    ax.collections[0].colorbar.set_label("accuracy")
    ax.set(ylabel='$'  + character + '$')
    plt.title(acc_desc + ' w.r.t $' + character + '$')
    #sns.set_style("whitegrid", {'axes.grid' : False})
    b, t = plt.ylim()
    b += 0.5
    t -= 0.5
    custom_ylim = (b, t)
    plt.setp(ax, ylim=custom_ylim)
    plt.show()
```

Trial 1

In [29]:
```
# To store results of models in trial 1
t1_accu_dict = {}

t1_train_dict = {}
t1_test_dict = {}

t1_optimal_train_dict = {}
t1_optimal_test_dict = {}
```

In [30]:
```
from sklearn.model_selection import train_test_split
# split the data into testing and training sets
X1_train, X1_test, y1_train, y1_test = train_test_split(X, y, train_siz
e = 0.8, test_size=0.20, random_state=42)

print("Shape of input data X_train: {} and shape of target variable y_t
rain: {}".format(X1_train.shape, y1_train.shape))
print("Shape of input data X_test: {} and shape of target variable y_te
st: {}".format(X1_test.shape, y1_test.shape))
```

Accuracy Performance

Model 'refit' set to accuracy

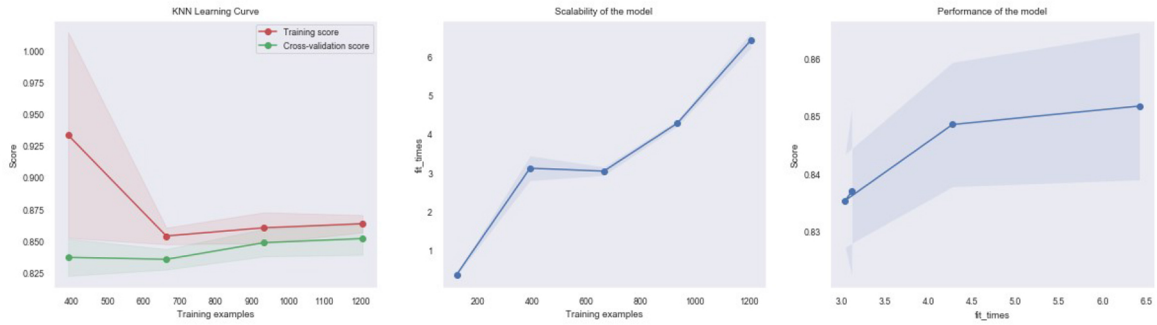
In [31]:
```
%%time
# run the train_algo function and train the data on the models
train_algo(gridcvs, X1_train, y1_train, 5)
```

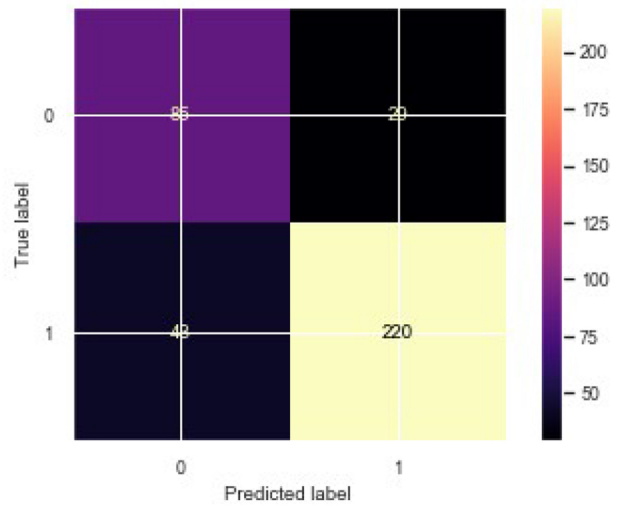
```
outer fold 1/5 | tuning DecisionTree | inner ACC 90.64% | outer ACC 7
2.19%
outer fold 1/5 | tuning KNN      | inner ACC 85.13% | outer ACC 82.4
5%
outer fold 1/5 | tuning Logistic | inner ACC 85.23% | outer ACC 82.7
8%
outer fold 1/5 | tuning RandomForest | inner ACC 83.60% | outer ACC 8
1.79%
outer fold 2/5 | tuning DecisionTree | inner ACC 90.55% | outer ACC 6
8.54%
outer fold 2/5 | tuning KNN      | inner ACC 85.12% | outer ACC 81.4
6%
outer fold 2/5 | tuning Logistic | inner ACC 85.12% | outer ACC 84.1
1%
outer fold 2/5 | tuning RandomForest | inner ACC 83.38% | outer ACC 8
2.45%
outer fold 3/5 | tuning DecisionTree | inner ACC 90.41% | outer ACC 6
6.56%
outer fold 3/5 | tuning KNN      | inner ACC 85.47% | outer ACC 78.4
8%
outer fold 3/5 | tuning Logistic | inner ACC 85.72% | outer ACC 78.8
1%
outer fold 3/5 | tuning RandomForest | inner ACC 84.66% | outer ACC 7
7.81%
outer fold 4/5 | tuning DecisionTree | inner ACC 89.71% | outer ACC 7
0.10%
outer fold 4/5 | tuning KNN      | inner ACC 85.30% | outer ACC 79.0
7%
outer fold 4/5 | tuning Logistic | inner ACC 85.13% | outer ACC 80.0
7%
outer fold 4/5 | tuning RandomForest | inner ACC 84.12% | outer ACC 7
8.41%
outer fold 5/5 | tuning DecisionTree | inner ACC 90.45% | outer ACC 7
0.43%
outer fold 5/5 | tuning KNN      | inner ACC 85.42% | outer ACC 79.4
0%
outer fold 5/5 | tuning Logistic | inner ACC 85.42% | outer ACC 80.7
3%
outer fold 5/5 | tuning RandomForest | inner ACC 84.32% | outer ACC 8
0.73%
KNN      | outer CV acc. 80.17% +\- 1.518
RandomForest | outer CV acc. 80.24% +\- 1.831
Logistic | outer CV acc. 81.30% +\- 1.904
DecisionTree | outer CV acc. 69.56% +\- 1.897

KNN best parameters {'classifier__n_neighbors': 75, 'classifier__weig
hts': 'distance'}
RandomForest best parameters {'classifier__max_features': 12}
Logistic best parameters {'classifier__C': 1.0, 'classifier__penalty
': 'l2'}
DecisionTree best parameters {'classifier__max_depth': 1}
Wall time: 2min 39s
```

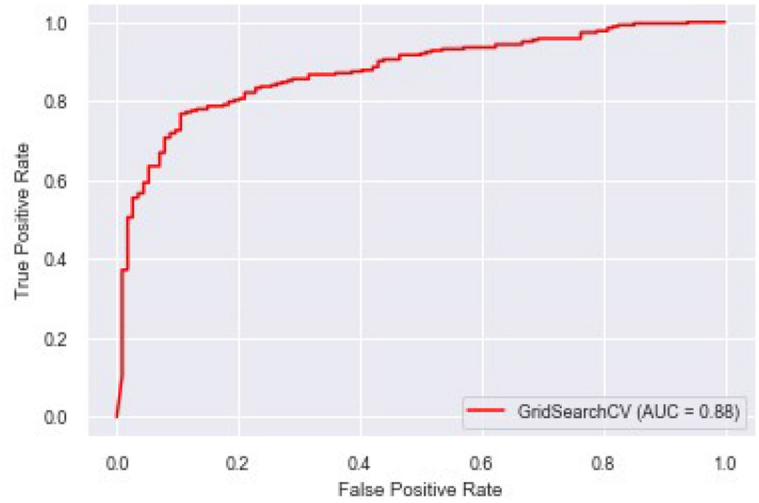
In [32]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['KNN'], 'KNN Learning Curve', X1_train, y1_
train);
# run the fit algo function and fit all the training data to KNN
fit_algo(t1_algo('KNN'), 'KNN', X1_train, y1_train, X1_test, y1_test, t
1_accu_dict,'KNN Accuracy', t1_train_dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['KNN'], X1_test, y1_test, cmap = 'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['KNN'], X1_test, y1_test, c ='red');
```

```
Accuracy 81.30% (average over CV test folds)
Training Accuracy: 100.00%
Test Accuracy: 79.58%
Wall time: 1min 35s
```

Out[32]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a60f56a188>
```



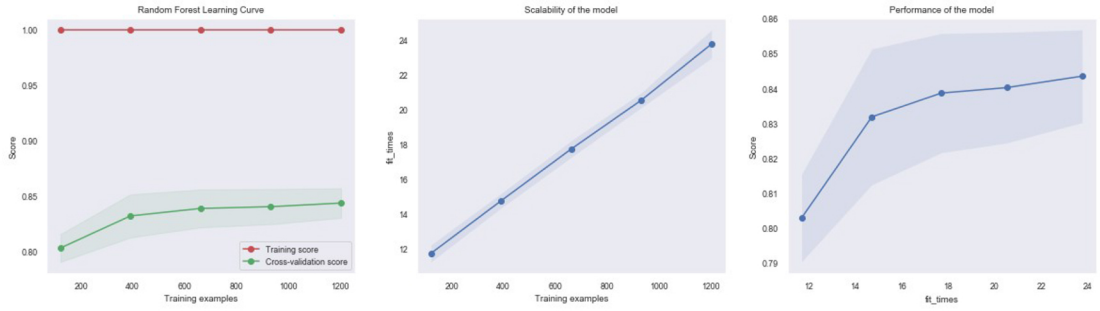


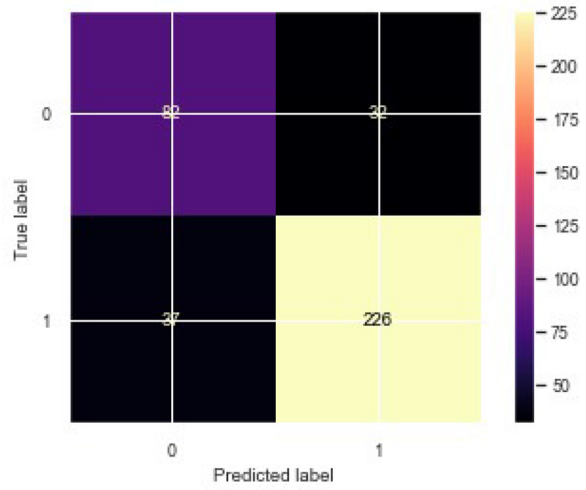


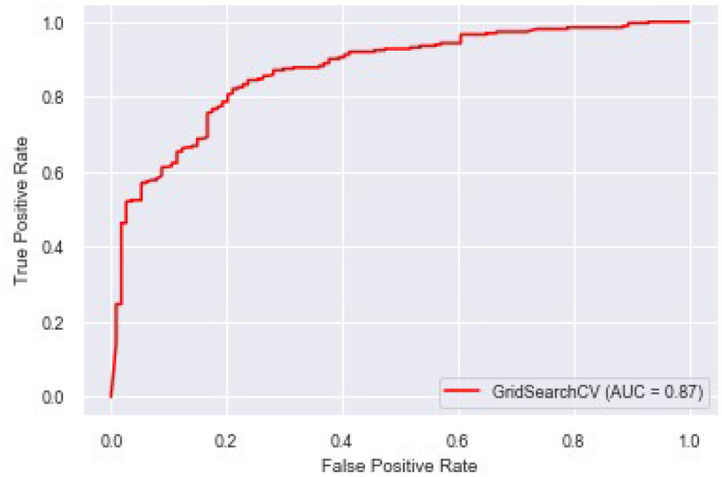
In [33]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['RandomForest'], 'Random Forest Learning Cu
rve', X1_train, y1_train);
# run the fit algo function and fit all the training data to RF
fit_algo(t1_algo('RandomForest'), 'RandomForest', X1_train, y1_train, X
1_test, y1_test, t1_accu_dict,'RandomForest Accuracy', t1_train_dict, t
1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['RandomForest'], X1_test, y1_test, cmap =
'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['RandomForest'], X1_test, y1_test, c ='red');
```

```
Accuracy 81.56% (average over CV test folds)
Training Accuracy: 100.00%
Test Accuracy: 80.11%
Wall time: 7min 54s
```

Out[33]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a60f682f88>



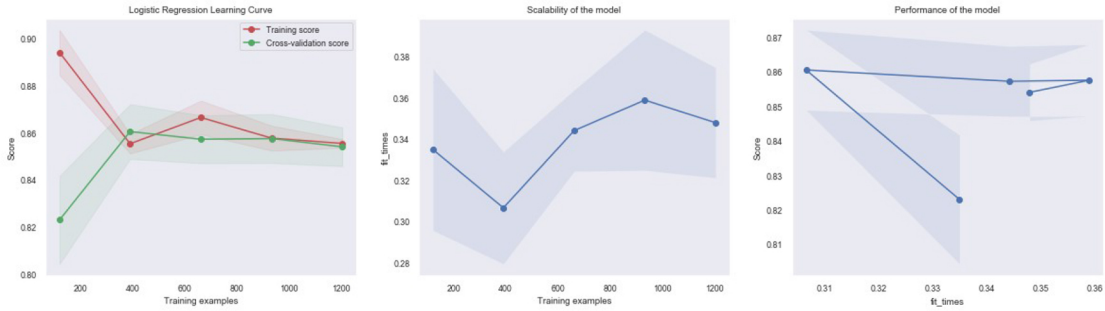


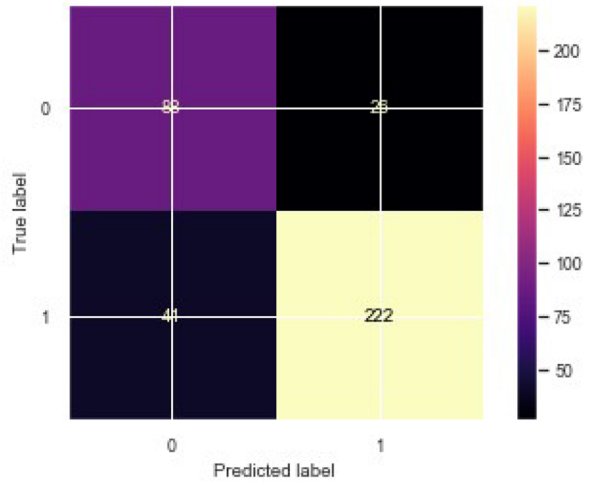


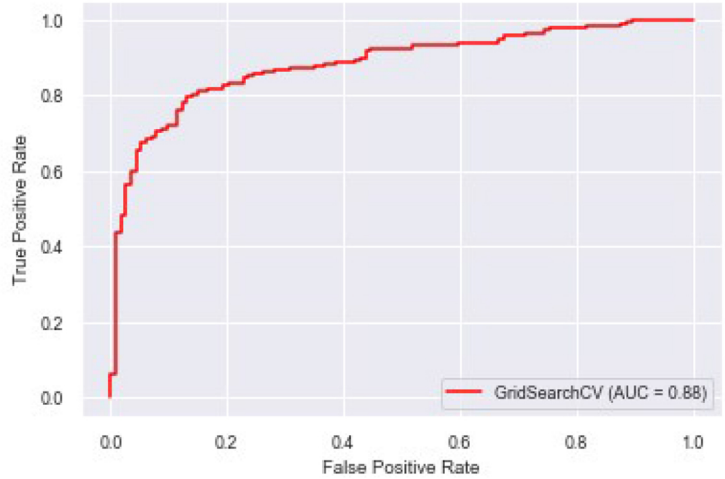
In [34]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['Logistic'], 'Logistic Regression Learning Curve', X1_train, y1_train);
# run the fit algo function and fit all the training data to Logistic Regression
fit_algo(t1_algo('Logistic'), 'Logistic', X1_train, y1_train, X1_test, y1_test, t1_accu_dict,'Logistic Accuracy', t1_train_dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['Logistic'], X1_test, y1_test, cmap = 'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['Logistic'], X1_test, y1_test, c ='red');
```

```
Accuracy 81.63% (average over CV test folds)
Training Accuracy: 81.83%
Test Accuracy: 80.37%
Wall time: 8.93 s
```

Out[34]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a60f7d5888>



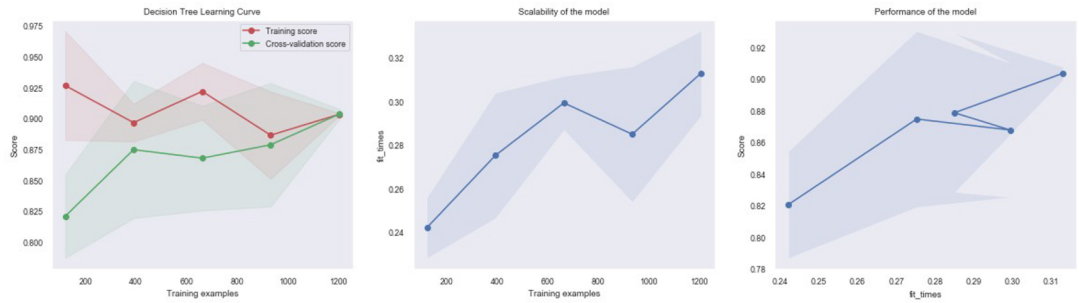


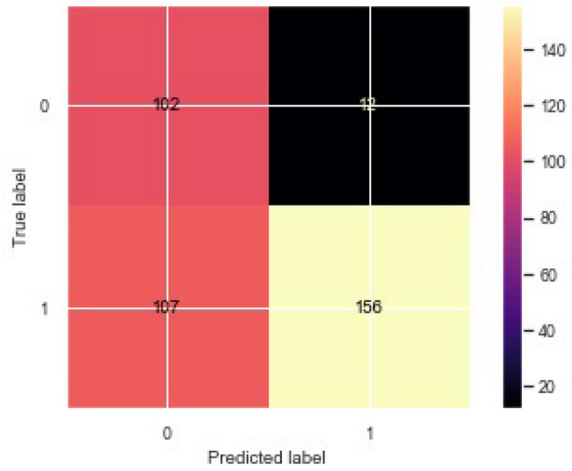


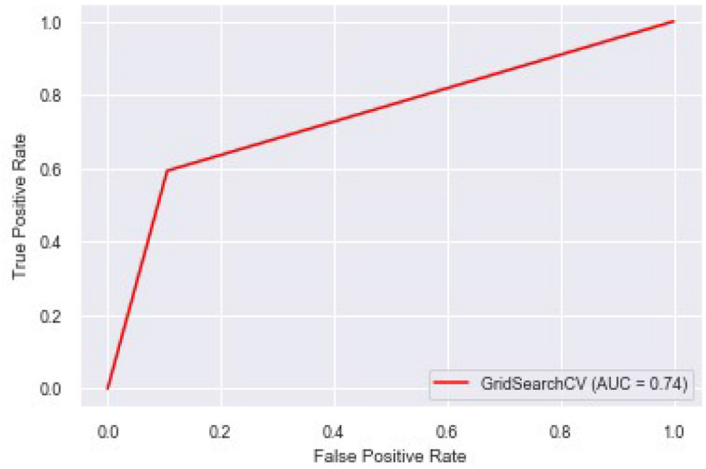
In [35]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['DecisionTree'], 'Decision Tree Learning Curve', X1_train, y1_train);
# run the fit algo function and fit all the training data to Decision Tree
fit_algo(t1_algo('DecisionTree'), 'DecisionTree', X1_train, y1_train, X1_test, y1_test, t1_accu_dict,'DecisionTree Accuracy', t1_train_dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['DecisionTree'], X1_test, y1_test, cmap = 'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['DecisionTree'], X1_test, y1_test, c ='red');
```

```
Accuracy 78.71% (average over CV test folds)
Training Accuracy: 81.43%
Test Accuracy: 77.45%
Wall time: 7.52 s
```

Out[35]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a6102cbf88>
```







In [37]:
```
%%time
# run the optimized function and apply optimal parameters to algorithms
and look at training/testing scores
optimized('distance', 75, 12, 1, 1, 'l2', X1_train, y1_train, X1_test,
y1_test,t1_optimal_train_dict,t1_optimal_test_dict)
```

```
KNN Train Accuracy 0.8010450815163583
KNN Test Accuracy 0.7771883289124668
              precision    recall  f1-score   support

           0       0.62      0.69      0.65       114
           1       0.86      0.81      0.84       263

    accuracy                           0.78       377
   macro avg       0.74      0.75      0.74       377
weighted avg       0.79      0.78      0.78       377

Random Forest Train Accuracy 0.8103298057248465
Random Forest Test Accuracy 0.8090185676392573
              precision    recall  f1-score   support

           0       0.67      0.73      0.70       114
           1       0.88      0.84      0.86       263

    accuracy                           0.81       377
   macro avg       0.77      0.79      0.78       377
weighted avg       0.81      0.81      0.81       377

Logistic Train Accuracy 0.8156212184550394
Logistic Test Accuracy 0.8010610079575596
              precision    recall  f1-score   support

           0       0.68      0.66      0.67       114
           1       0.85      0.86      0.86       263

    accuracy                           0.80       377
   macro avg       0.76      0.76      0.76       377
weighted avg       0.80      0.80      0.80       377

Decision Tree Train Accuracy 0.78645134320477
Decision Tree Test Accuracy 0.7745358090185677
              precision    recall  f1-score   support

           0       0.61      0.72      0.66       114
           1       0.87      0.80      0.83       263

    accuracy                           0.77       377
   macro avg       0.74      0.76      0.75       377
weighted avg       0.79      0.77      0.78       377

Wall time: 11.2 s
```

Precision Performance

Model 'refit' set to precision

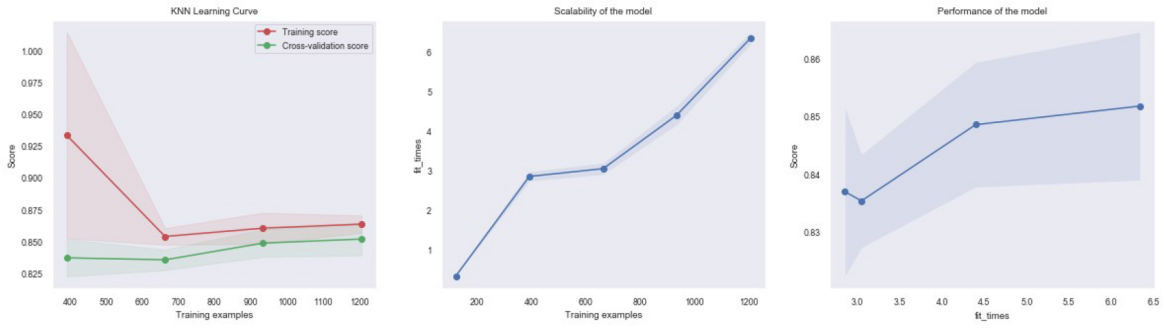
[38]:
```
train_algo_prec(gridcvs, X1_train, y1_train, 5)
```

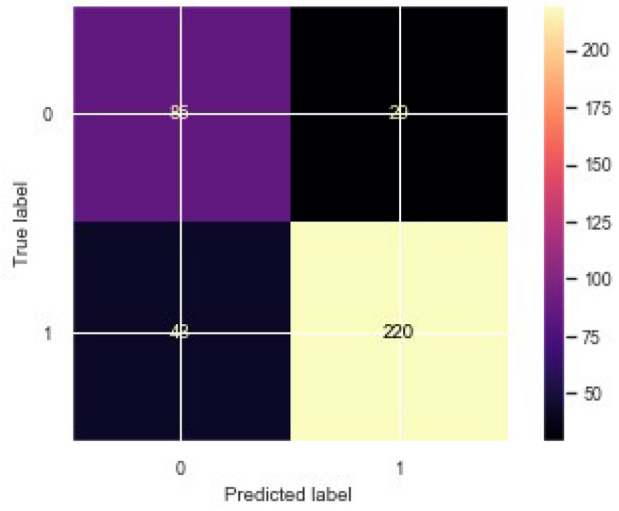
```
outer fold 1/5 | tuning DecisionTree | inner prec 90.64% | outer prec
89.80%
outer fold 1/5 | tuning KNN      | inner prec 85.13% | outer prec 87.
00%
outer fold 1/5 | tuning Logistic | inner prec 85.23% | outer prec 85.
31%
outer fold 1/5 | tuning RandomForest | inner prec 84.08% | outer prec
83.80%
outer fold 2/5 | tuning DecisionTree | inner prec 90.55% | outer prec
89.55%
outer fold 2/5 | tuning KNN      | inner prec 85.12% | outer prec 84.
04%
outer fold 2/5 | tuning Logistic | inner prec 85.12% | outer prec 84.
93%
outer fold 2/5 | tuning RandomForest | inner prec 83.32% | outer prec
83.78%
outer fold 3/5 | tuning DecisionTree | inner prec 90.41% | outer prec
89.60%
outer fold 3/5 | tuning KNN      | inner prec 85.47% | outer prec 86.
89%
outer fold 3/5 | tuning Logistic | inner prec 85.72% | outer prec 86.
17%
outer fold 3/5 | tuning RandomForest | inner prec 84.54% | outer prec
83.51%
outer fold 4/5 | tuning DecisionTree | inner prec 89.71% | outer prec
92.97%
outer fold 4/5 | tuning KNN      | inner prec 85.30% | outer prec 86.
63%
outer fold 4/5 | tuning Logistic | inner prec 85.13% | outer prec 87.
23%
outer fold 4/5 | tuning RandomForest | inner prec 84.08% | outer prec
85.20%
outer fold 5/5 | tuning DecisionTree | inner prec 90.45% | outer prec
89.93%
outer fold 5/5 | tuning KNN      | inner prec 85.42% | outer prec 83.
17%
outer fold 5/5 | tuning Logistic | inner prec 85.42% | outer prec 84.
47%
outer fold 5/5 | tuning RandomForest | inner prec 84.33% | outer prec
81.90%
KNN      | outer CV prec. 85.55% +\- 1.612
RandomForest | outer CV prec. 83.64% +\- 1.052
Logistic | outer CV prec. 85.62% +\- 0.981
DecisionTree | outer CV prec. 90.37% +\- 1.307

KNN best parameters {'classifier__n_neighbors': 75, 'classifier__weig
hts': 'distance'}
RandomForest best parameters {'classifier__max_features': 12}
Logistic best parameters {'classifier__C': 1.0, 'classifier__penalty
': 'l2'}
DecisionTree best parameters {'classifier__max_depth': 1}
```

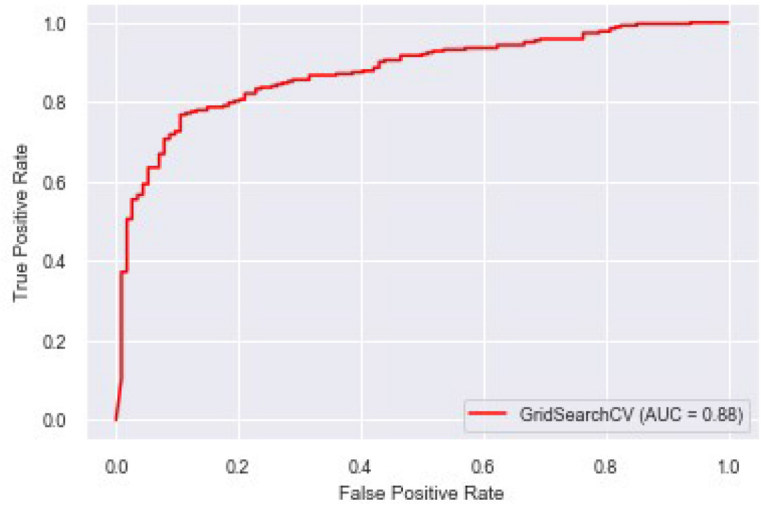
In [39]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['KNN'], 'KNN Learning Curve', X1_train, y1_
train);
# run the fit algo function and fit all the training data to KNN
fit_algo_prec(t1_algo('KNN'), 'KNN', X1_train, y1_train, X1_test, y1_te
st, t1_accu_dict,'KNN Precision', t1_train_dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['KNN'], X1_test, y1_test, cmap = 'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['KNN'], X1_test, y1_test, c ='red');
```

```
Precision 81.30% (average over CV test folds)
Training Precision: 100.00%
Test Precision: 87.50%
Wall time: 1min 35s
```

Out[39]: <sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a610226748>



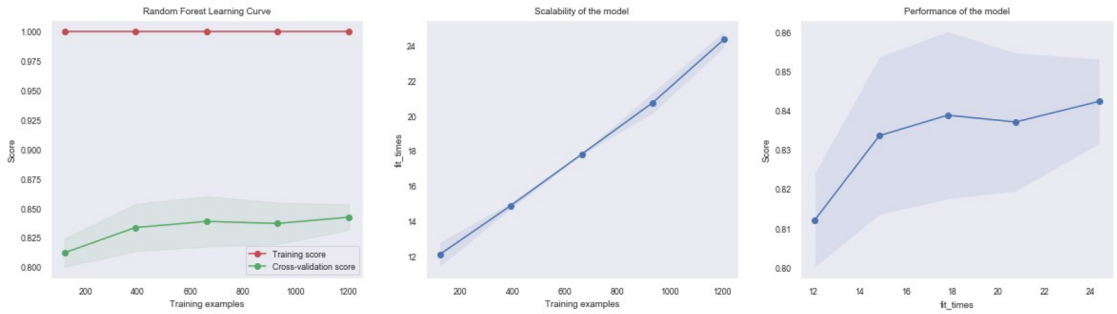


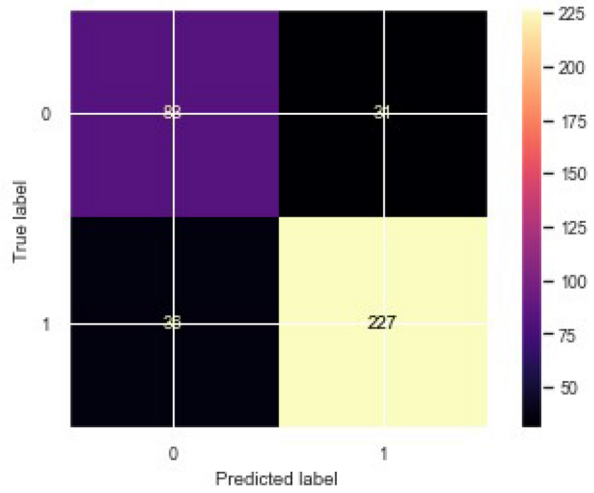


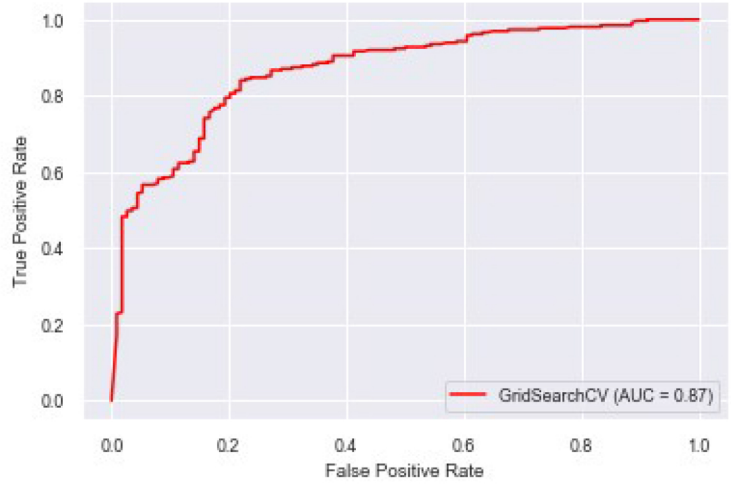
In [40]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['RandomForest'], 'Random Forest Learning Cu
rve', X1_train, y1_train);
# run the fit algo function and fit all the training data to KNN
fit_algo_prec(t1_algo('RandomForest'), 'RandomForest', X1_train, y1_tra
in, X1_test, y1_test, t1_accu_dict,'Random Forest Precision', t1_train_
dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['RandomForest'], X1_test, y1_test, cmap =
'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['RandomForest'], X1_test, y1_test, c ='red');
```

```
Precision 81.56% (average over CV test folds)
Training Precision: 100.00%
Test Precision: 87.75%
Wall time: 8min 3s
```

Out[40]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a6102aa648>
```



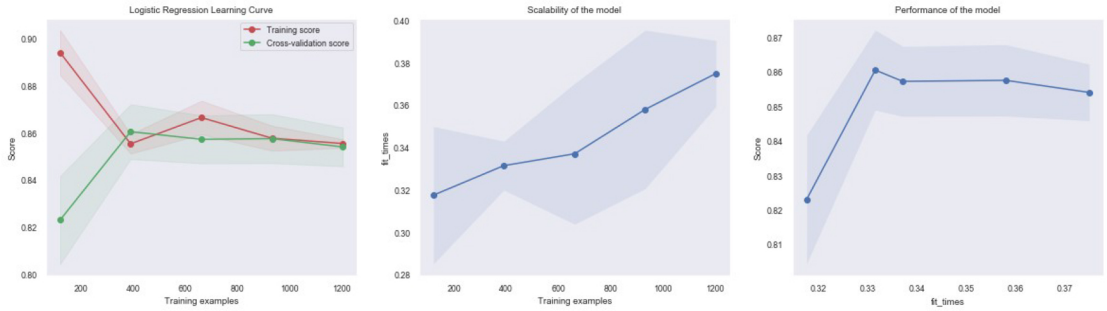


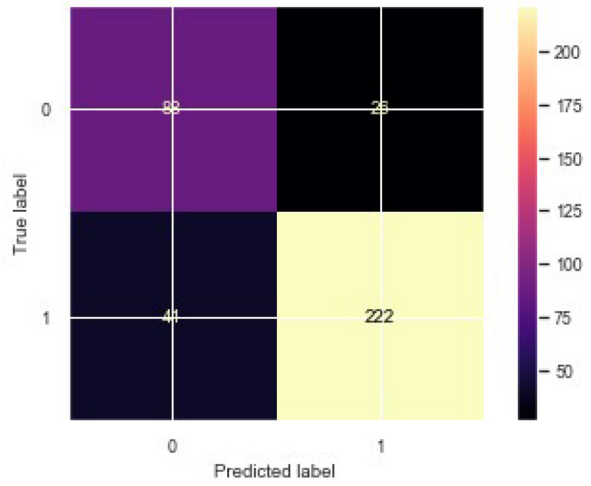


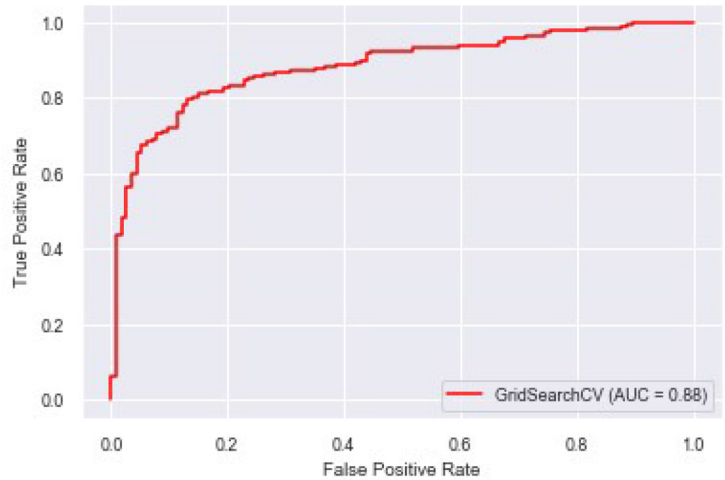
In [41]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['Logistic'], 'Logistic Regression Learning
Curve', X1_train, y1_train);
# run the fit algo function and fit all the training data to KNN
fit_algo_prec(t1_algo('Logistic'), 'Logistic', X1_train, y1_train, X1_t
est, y1_test, t1_accu_dict,'Logistic Precision', t1_train_dict, t1_test
_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['Logistic'], X1_test, y1_test, cmap = 'ma
gma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['Logistic'], X1_test, y1_test, c ='red');
```

```
Precision 81.63% (average over CV test folds)
Training Precision: 84.60%
Test Precision: 85.93%
Wall time: 9.09 s
```

Out[41]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a60f7bdbc8>
```



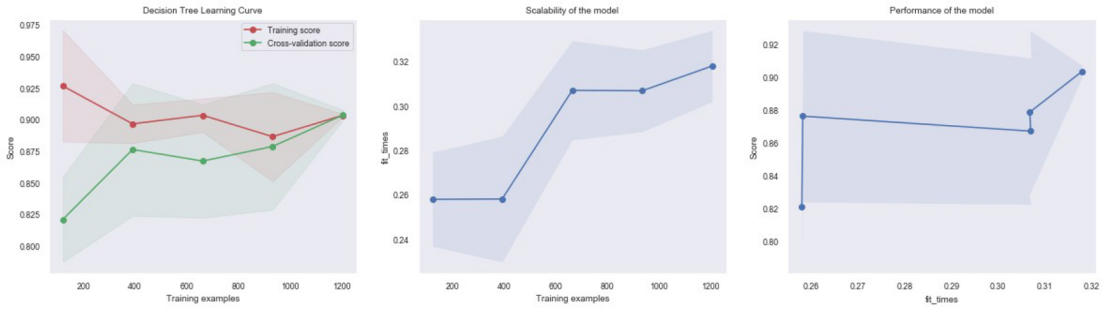


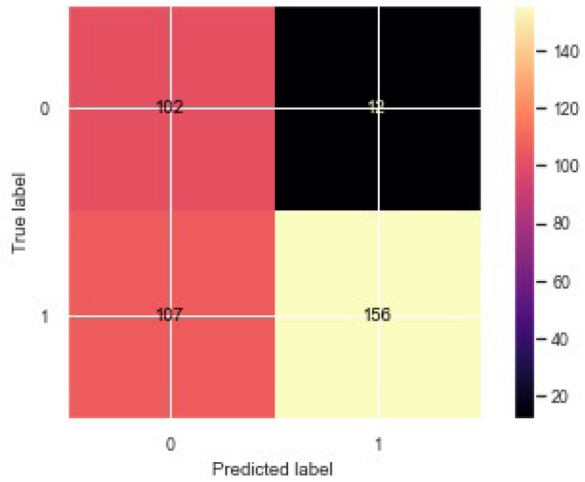


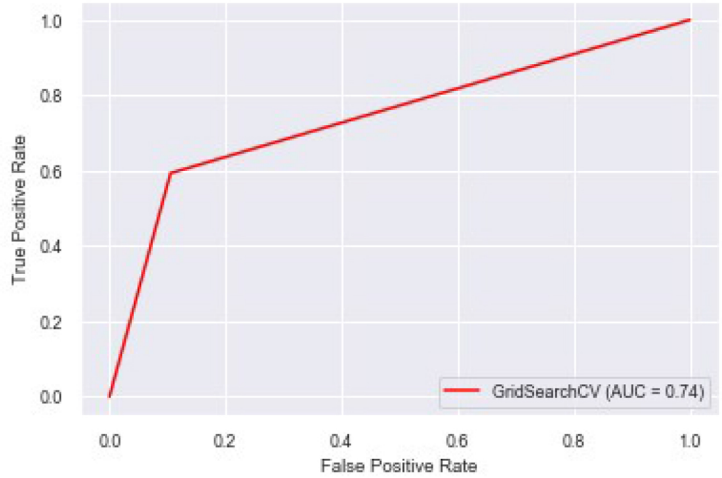
In [42]:
```
%%time
# plot learning curves for training
plot_learning_curve(gridcvs['DecisionTree'], 'Decision Tree Learning Curve', X1_train, y1_train);
# run the fit algo function and fit all the training data to KNN
fit_algo_prec(t1_algo('DecisionTree'), 'DecisionTree', X1_train, y1_train, X1_test, y1_test, t1_accu_dict,'Decision Tree Precision', t1_train_dict, t1_test_dict)
# plot confusion matrix on the test samples
plot_confusion_matrix(gridcvs['DecisionTree'], X1_test, y1_test, cmap = 'magma')
# plot ROC curve on the test samples
plot_roc_curve(gridcvs['DecisionTree'], X1_test, y1_test, c ='red');
```

```
Precision 78.58% (average over CV test folds)
Training Precision: 86.10%
Test Precision: 86.78%
Wall time: 7.7 s
```

Out[42]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x1a6189b2ec8>
```







In [43]:
```
optimized_prec('uniform', 75, 12, 1, 1, 'l2', X1_train, y1_train, X1_test, y1_test,t1_optimal_train_dict,t1_optimal_test_dict)
```

```
KNN Train Precision 0.857566765578635
KNN Test Precision 0.8870967741935484
              precision    recall  f1-score   support

           0       0.67      0.75      0.71       114
           1       0.89      0.84      0.86       263

    accuracy                           0.81       377
   macro avg       0.78      0.80      0.78       377
weighted avg       0.82      0.81      0.81       377

Random Forest Train Precision 1.0
Random Forest Test Precision 0.8844621513944223
              precision    recall  f1-score   support

           0       0.67      0.75      0.71       114
           1       0.88      0.84      0.86       263

    accuracy                           0.81       377
   macro avg       0.78      0.79      0.79       377
weighted avg       0.82      0.81      0.82       377

Logistic Train Precision 0.8553149606299213
Logistic Test Precision 0.8906882591093117
              precision    recall  f1-score   support

           0       0.67      0.76      0.71       114
           1       0.89      0.84      0.86       263

    accuracy                           0.81       377
   macro avg       0.78      0.80      0.79       377
weighted avg       0.82      0.81      0.82       377

Decision Tree Train Precision 0.9034175334323923
Decision Tree Test Precision 0.9285714285714286
              precision    recall  f1-score   support

           0       0.49      0.89      0.63       114
           1       0.93      0.59      0.72       263

    accuracy                           0.68       377
   macro avg       0.71      0.74      0.68       377
weighted avg       0.80      0.68      0.70       377
```

Results: Trial 1

In [44]:
```
t1_df_accu_dict = pd.DataFrame.from_dict(t1_accu_dict,orient = "index",
columns=["Score"]).sort_values(by=["Score"], ascending=False)
t1_df_train_dict = pd.DataFrame.from_dict(t1_train_dict,orient = "inde
x",columns=["Score"]).sort_values(by=["Score"], ascending=False)
t1_df_test_dict = pd.DataFrame.from_dict(t1_test_dict,orient = "index",
columns=["Score"]).sort_values(by=["Score"], ascending=False)
t1_df_optimal_train_dict = pd.DataFrame.from_dict(t1_optimal_train_dic
t,orient = "index",columns=["Score"]).sort_values(by=["Score"], ascendi
ng=False)
t1_df_optimal_test_dict = pd.DataFrame.from_dict(t1_optimal_test_dict,o
rient = "index",columns=["Score"]).sort_values(by=["Score"], ascending=
False)
```

In [45]:
```
print("Average Accuracy Across Folds")
print(t1_df_accu_dict)
print("")
print("Training Scores")
print(t1_df_train_dict)
print("")
print("Testing Scores")
print(t1_df_test_dict)
print("")
print("Training Scores For Optimal Parameter")
print(t1_df_optimal_train_dict)
print("")
print("Testing Scores For Optimal Parameter")
print(t1_df_optimal_test_dict)
```

Average Accuracy Across Folds

	Score
Logistic Accuracy	0.816283
Logistic Precision	0.816283
Random Forest Precision	0.815630
RandomForest Accuracy	0.815628
KNN Accuracy	0.812979
KNN Precision	0.812979
DecisionTree Accuracy	0.787116
Decision Tree Precision	0.785787

Training Scores

	Score
KNN Training Accuracy	1.000000
RandomForest Training Accuracy	1.000000
KNN Train Precision	1.000000
RandomForest Train Precision	1.000000
DecisionTree Train Precision	0.861000
Logistic Train Precision	0.846008
Logistic Training Accuracy	0.818302
DecisionTree Training Accuracy	0.814324

Testing Scores

	Score
RandomForest Test Precision	0.877470
KNN Test Precision	0.875000
DecisionTree Test Precision	0.867769
Logistic Test Precision	0.859316
Logistic Test Accuracy	0.803714
RandomForest Test Accuracy	0.801061
KNN Test Accuracy	0.795756
DecisionTree Test Accuracy	0.774536

Training Scores For Optimal Parameter

	Score
Random Forest Train Precision	1.000000
Decision Tree Train Precision	0.903418
KNN Train Precision	0.857567
Logistic Train Precision	0.855315
Logistic Train Accuracy	0.815621
Random Forest Train Accuracy	0.810330
KNN Train Accuracy	0.801045
Decision Tree Train Accuracy	0.786451

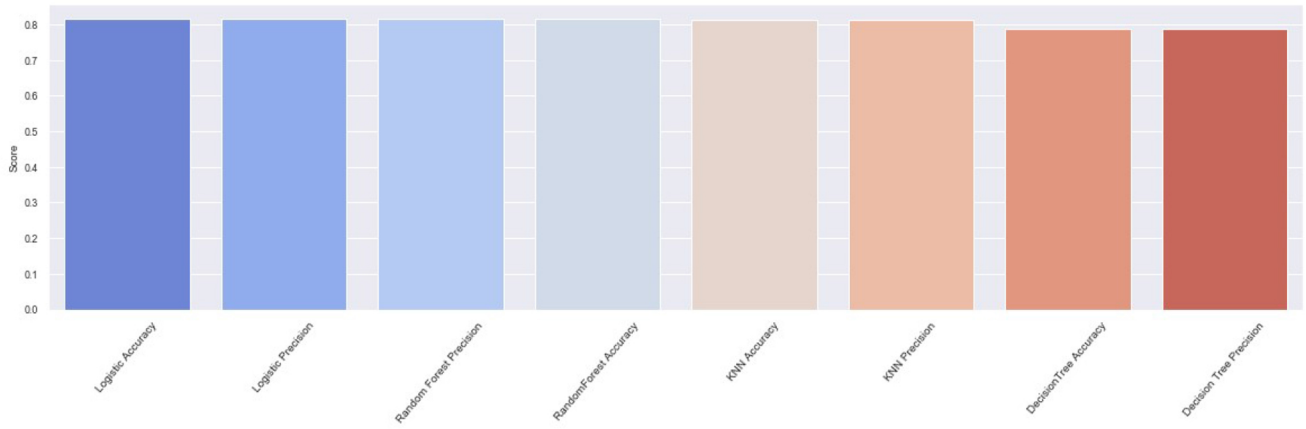
Testing Scores For Optimal Parameter

	Score
Decision Tree Test Precision	0.928571
Logistic Test Precision	0.890688
KNN Test Precision	0.887097
Random Forest Test Precision	0.884462
Random Forest Test Accuracy	0.809019
Logistic Test Accuracy	0.801061
KNN Test Accuracy	0.777188
Decision Tree Test Accuracy	0.774536

Visualizations

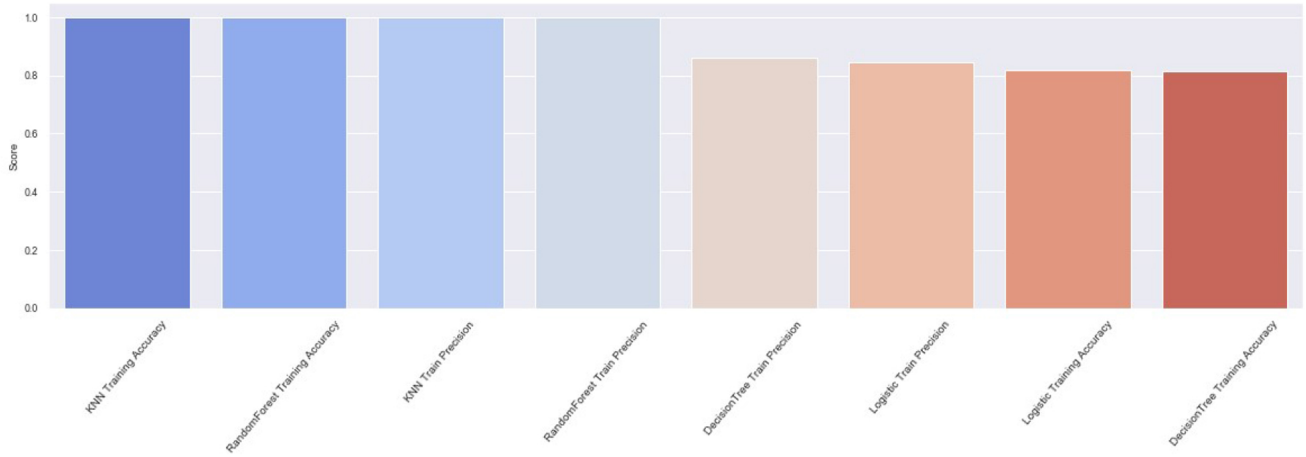
Average Accuracies Across Folds

In [46]:
```
fig = plt.figure(figsize = (20,5))

sns.barplot(x = t1_df_accu_dict.index,y = t1_df_accu_dict.Score,
            palette = 'coolwarm').set_xticklabels(t1_df_accu_dict.inde
x,rotation = 50, fontsize = 10)

plt.show()
```



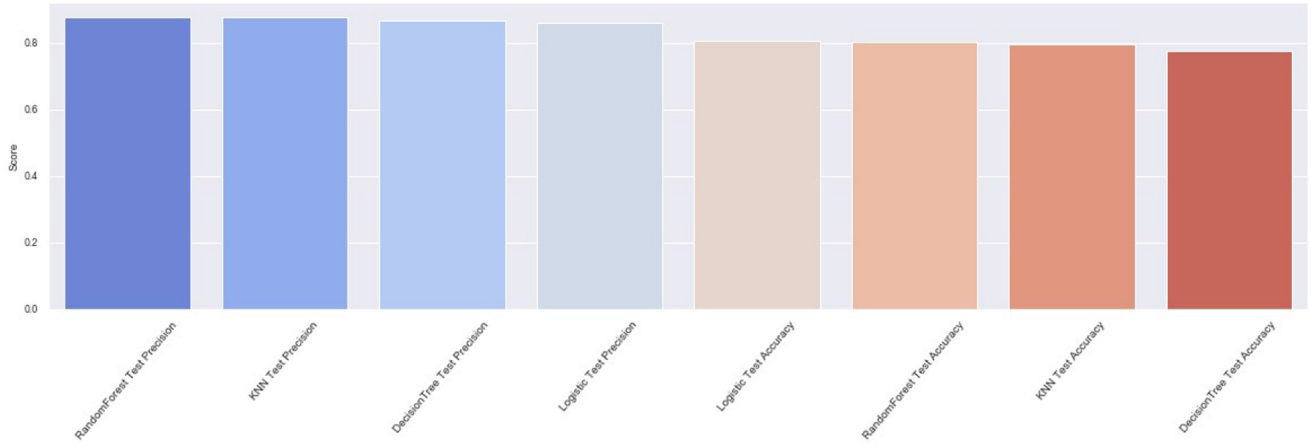
Algorithm Training Scores

In [47]:
```
fig = plt.figure(figsize = (20,5))

sns.barplot(x = t1_df_train_dict.index,y = t1_df_train_dict.Score,
            palette = 'coolwarm').set_xticklabels(t1_df_train_dict.inde
x,rotation = 50, fontsize = 10)

plt.show()
```



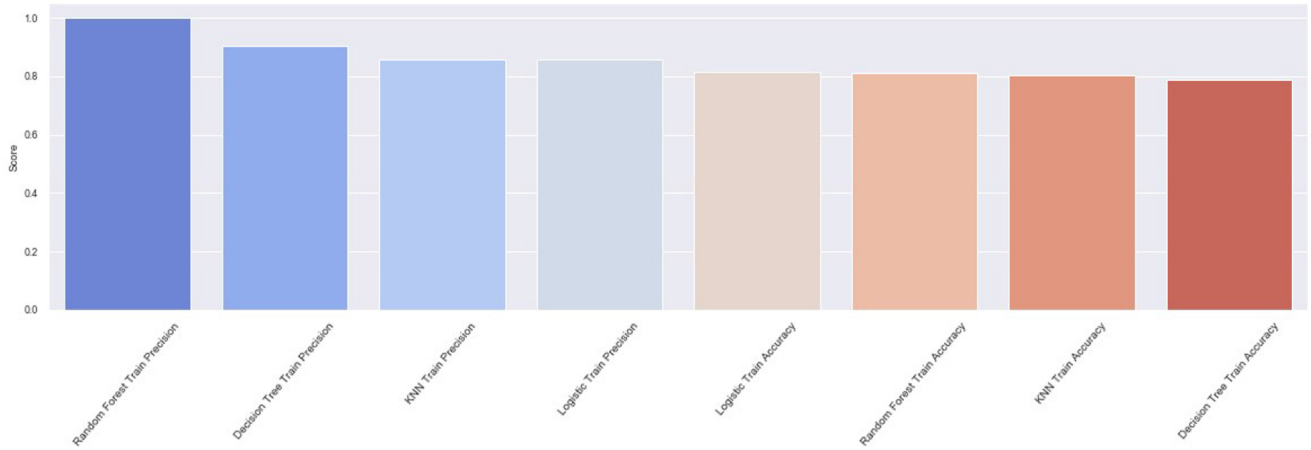
Algorithm Testing Scores

In [48]:
```
fig = plt.figure(figsize = (20,5))

sns.barplot(x = t1_df_test_dict.index,y = t1_df_test_dict.Score,
            palette = 'coolwarm').set_xticklabels(t1_df_test_dict.inde
x,rotation = 50, fontsize = 10)

plt.show()
```



Algorithm Training Scores With Optimal Hyperparameter

In [49]:
```
fig = plt.figure(figsize = (20,5))

sns.barplot(x = t1_df_optimal_train_dict.index,y = t1_df_optimal_train_
dict.Score,
            palette = 'coolwarm').set_xticklabels(t1_df_optimal_train_d
ict.index,rotation = 50, fontsize = 10)

plt.show()
```



Algorithm Testing Scores With Optimal Hyperparameter

In [50]:
```
fig = plt.figure(figsize = (20,5))

sns.barplot(x = t1_df_optimal_test_dict.index,y = t1_df_optimal_test_di
ct.Score,
            palette = 'coolwarm').set_xticklabels(t1_df_optimal_test_di
ct.index,rotation = 50, fontsize = 10)

plt.show()
```