Heart Disease Classification

Phase 3 Project by Bella Scribner

- Flex
- Instructor: Morgan Jones
- Date of Project Review: 1/8/24
- Blog: https://datascienceprojectsandmore.blogspot.com/ (https://datascienceprojectsandmore.blogspot.com/)

Introduction

Overview ¶

This project creates a binary classification machine learning model to predict whether or not a patient has heart disease. The project goes through a data exploration and cleaning process. Pipelines are utilized for both preprocessing and data as well as for iterating through different types of models. After tuning the two models with highest performance a final model is chosen and evaluated. Lastly, recommendations pulled from the model are given.

The Business + Business Problem

Heart Disease is a leading cause of death not only in the United States, but globally as well according to the World Health Organization and Center for Disease Control. Cardiovascular diseases are estimated to be the cause of death for 17.9 million lives each year globally.

This project is framed by a health care provider who would like to prevent further deaths by heart diseases with early identification and treatment. A binary classification machine learning model is created that takes in patients' information and predicts whether or not heart disease is present. The upmost goal is to identify as many people as possible who are ill prior to any serious health crises , such as a heart attack or stroke. Missing someone could be potentially fatal. However, the health care providers are also concerned with falsely categorizing healthy people with heart disease as this can place an undue burden on a healthy patient.

The Data Source

The data set used in this project, "Heart Failure Prediction Dataset," can be found on Kaggle (https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction/). The data is a combination of 5 separate datasets from the UCI Machine Learning Repository under the index of heart disease datasets.

In [1]:

```
import pandas as pd
import numpy as np
import scipy.stats as stats

from sklearn.model_selection import train_test_split, cross_val_score, Gri
RandomizedSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.inspection import permutation_importance

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, reca
accuracy_score, precision_score, f1_score, classification_report

# from imblearn.over_sampling import SMOTE
# from imblearn.pipeline import Pipeline as ImPipeline

import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
plt.style.use('bmh')
```

Data Exploration

First, let's take a look at what data we are working with.

In [2]:

```
data = pd.read_csv("heart.csv")
data.head()
```

Out[2]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	Exe
0	40	M	ATA	140	289	0	Normal	172	
1	49	F	NAP	160	180	0	Normal	156	
2	37	M	ATA	130	283	0	ST	98	
3	48	F	ASY	138	214	0	Normal	108	
4	54	M	NAP	150	195	0	Normal	122	

In [3]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Age             918 non-null    int64  
 1   Sex             918 non-null    object 
 2   ChestPainType   918 non-null    object 
 3   RestingBP       918 non-null    int64  
 4   Cholesterol     918 non-null    int64  
 5   FastingBS       918 non-null    int64  
 6   RestingECG      918 non-null    object 
 7   MaxHR           918 non-null    int64  
 8   ExerciseAngina  918 non-null    object 
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object 
 11  HeartDisease    918 non-null    int64  
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

Ideally, we wanted approximately 1,000 rows for this project; 918 data points is close enough, but we will have to be careful to keep as much data as possible.

Below is a more in depth description of all the variables we will be working with throughout the course of this project.

Column	Description
Age	Age of the patient
Sex	Sex of the patient
ChestPainType	Chest Pain Type: Typical Angina (TA), Atypical Angina (ATA), Non-Anginal Pain (NAP), or Asymptomatic (ASY)
RestingBP	Resting blood pressure (mm Hg)
Cholesterol	Serum Cholesterol (mm/dl)
FastingBS	Fasting blood sugar: 1 if FastingBS > 120 mg/dl, otherwise 0
RestingECG	Resting electrocardiogram results: Normal (Normal), Having ST-T wave abnormality (ST), or Showing probable or definite left ventricular hypertrophy by Estes' criteria (LVH)
MaxHR	Maximum heart rate achieved
ExerciseAngina	Yes if exercise-induced angina is present, otherwise No
Oldpeak	ST depression ('ST' relates to positions on the ECG plot)
ST_Slope	The slope of the peak exercise ST segment: Upsloping (Up), Flat (Flat), or Downsloping (Down)
HeartDisease	Target: 1 if the patient has heart disease, otherwise 0

In [4]:
```
# Missing Values Check
data.isna().sum()
```

Out[4]:
```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
HeartDisease      0
dtype: int64
```

In [5]:
```
# Check basic statistics of numerical variables
data.describe()
```

Out[5]:

	Age	RestingBP	Cholesterol	FastingBS	MaxHR	Oldpeak	HeartDisea
count	918.000000	918.000000	918.000000	918.000000	918.000000	918.000000	918.0000
mean	53.510893	132.396514	198.799564	0.233115	136.809368	0.887364	0.5533
std	9.432617	18.514154	109.384145	0.423046	25.460334	1.066570	0.4974
min	28.000000	0.000000	0.000000	0.000000	60.000000	-2.600000	0.0000
25%	47.000000	120.000000	173.250000	0.000000	120.000000	0.000000	0.0000
50%	54.000000	130.000000	223.000000	0.000000	138.000000	0.600000	1.0000
75%	60.000000	140.000000	267.000000	0.000000	156.000000	1.500000	1.0000
max	77.000000	200.000000	603.000000	1.000000	202.000000	6.200000	1.0000

From the above, we can see that most data points have a Fasting Blood Sugar below the cutoff of 120 mg/dl. More notably, we can see that there are some values of 0 for both Cholesterol and Resting Blood Pressure. We will assume that a zero value input is in place of a missing value (as having a resting blood pressure or cholesterol of zero is not possible). Let's look at the data points with these zero values.

In [6]:
```
data[data['RestingBP'] == 0]
```

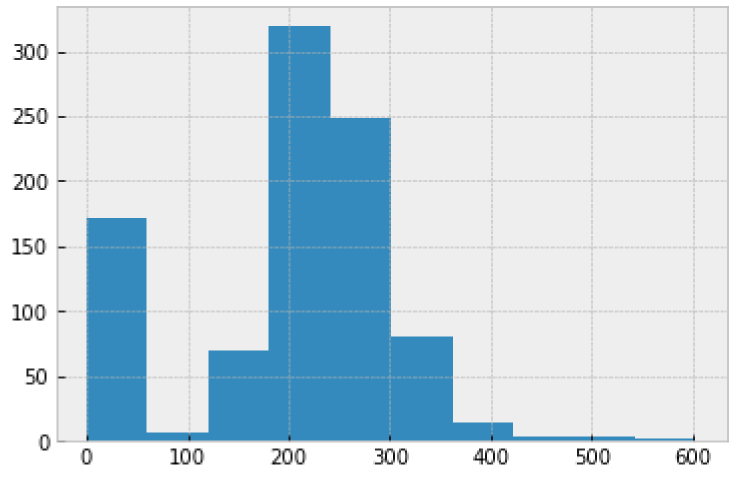
Out[6]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	E
449	55	M	NAP	0	0	0	Normal	155	

Seeing as there is only one entry with a zero Resting Blood Pressure value, we will move forward with dropping this entry. Next, we will examine the datapoints with zero values inputted for cholesterol.

In [7]:
```
data = data[data['RestingBP'] != 0]
```

In [8]:
```
data['Cholesterol'].hist();
```



In [9]:
```
data[data['Cholesterol'] == 0]
```

Out[9]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	E
293	65	M	ASY	115	0	0	Normal	93	
294	32	M	TA	95	0	1	Normal	127	
295	61	M	ASY	105	0	1	Normal	110	
296	50	M	ASY	145	0	1	Normal	139	
297	57	M	ASY	110	0	1	ST	131	
...	...	...	...	...	...	...	...	...	
514	43	M	ASY	122	0	0	Normal	120	
515	63	M	NAP	130	0	1	ST	160	
518	48	M	NAP	102	0	1	ST	110	
535	56	M	ASY	130	0	0	LVH	122	
536	62	M	NAP	133	0	1	ST	119	

171 rows × 12 columns

In [10]:
```
data[data['Cholesterol'] == 0]['HeartDisease'].value_counts()
```

Out[10]:
```
1    151
0     20
Name: HeartDisease, dtype: int64
```

That is a significant number of entries! And, a majority are entries where the patient does indeed have heart disease. Seeing as we do not want to lose these data points, we will replace all zero values with a NaN, and later input the mean cholesterol for these entries.

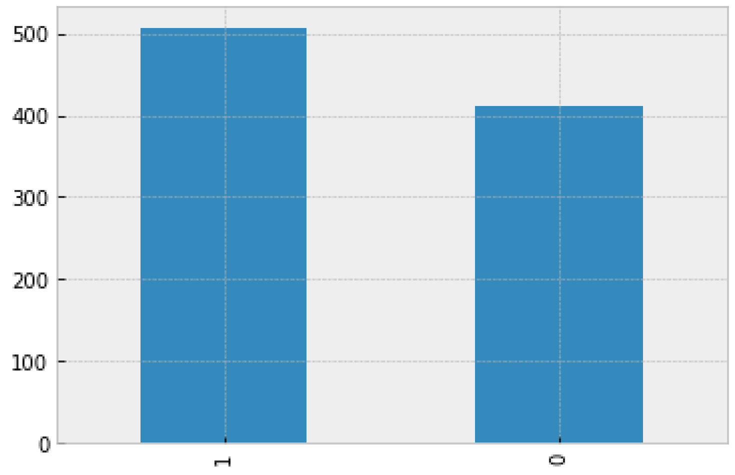
In [11]:
```
data.replace({'Cholesterol': {0 :np.nan}}, inplace=True)
```

In [12]:
```
data.isna().sum()
```

Out[12]:
```
Age                 0
Sex                 0
ChestPainType       0
RestingBP           0
Cholesterol       171
FastingBS           0
RestingECG          0
MaxHR               0
ExerciseAngina      0
Oldpeak             0
ST_Slope            0
HeartDisease        0
dtype: int64
```

Lastly, let's take a closer look at our target variable.

In [13]:
```
data['HeartDisease'].value_counts().plot.bar();
```



In [14]:
```
data['HeartDisease'].value_counts(normalize=True)
```

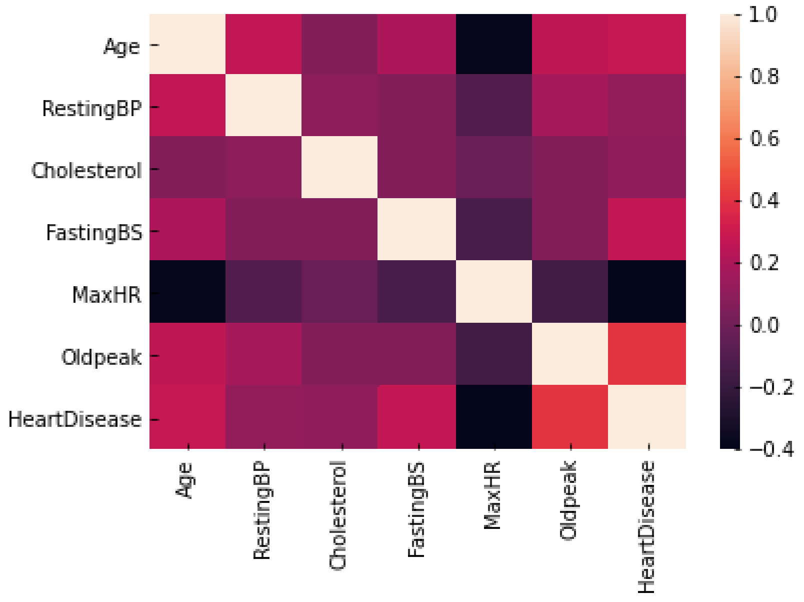
Out[14]:
```
1    0.55289
0    0.44711
Name: HeartDisease, dtype: float64
```

We have a slight imbalance between the two target categories, but not one that requires any sort of data augmentation such as SMOTE. Based on this distribution as well as the goals of the project, we will want to look at more than just accuracy when evaluating the models. Recall will most definitely be an important metric to evaluate our models.

Next, a peek at correlation with a heat map. From there we will look at the distribution of the target variable in relation to a few of the independent variables.

In [15]:

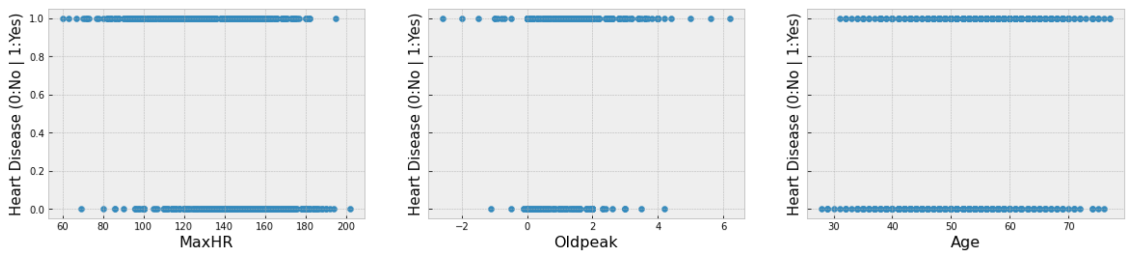
```
sns.heatmap(data.corr());
```



From the above heatmap of Pearson's correlation coefficients, we can see that none of the independent variables are highly correlated with each other. Furthermore, the three variables that are most correlated with the dependent variable are Max Heart Rate, Oldpeak and Age. Let's take a look at the distribution of our target against each of these three variables.

In [16]:

```
fig, ax = plt.subplots(ncols=3, sharey=True, figsize=(20,4))
i=0
for var in ['MaxHR', 'Oldpeak', 'Age']:
    a = i % 3
    ax[a].scatter(data['{}'.format(var)], data['HeartDisease'])
    ax[a].set_xlabel('{}'.format(var), fontsize=16)
    ax[a].set_ylabel('Heart Disease (0:No | 1:Yes)', fontsize=15)
    i+=1;
```



Data Pre-Processing

Now that we have a better understanding of the data and its components, we can begin cleaning and prepping the data for modeling.

In [17]:
```
# Split data
X = data.drop('HeartDisease', axis=1)
y = data['HeartDisease']
X_train_all, X_test, y_train_all, y_test = train_test_split(X, y, random_s
```

In [18]:
```
# Create Validation set
X_train, X_val, y_train, y_val = train_test_split(X_train_all, y_train_all
```

In [19]:
```
X_train.head()
```

Out[19]:

	Age	Sex	ChestPainType	RestingBP	Cholesterol	FastingBS	RestingECG	MaxHR	E
661	49	M	NAP	118	149.0	0	LVH	126	
645	57	M	NAP	128	229.0	0	LVH	150	
190	46	M	ASY	180	280.0	0	ST	120	
415	66	F	ASY	155	NaN	1	Normal	90	
188	50	F	ASY	120	328.0	0	Normal	110	

In [20]:
```
# Seperate Categorical variables in preperation of encoding
# We will include FastingBS as this is an over/under cutt off value
cat_cols = ['Sex', 'ChestPainType', 'FastingBS', 'RestingECG', 'ExerciseAn

# Encoding of Categorical Data
X_encoded = ColumnTransformer(transformers = [("ohe", OneHotEncoder(handle
                                                cat_cols)], remainder =
```

In [21]:
```
# Create the pipeline that will include all preprocessing steps:
    # 1. Encode the categorical data
    # 2. Inpute the mean for all missing values (cholesterol)
    # 3. Scale the data
preprocess_pipe = Pipeline(steps = [
    ('encode', X_encoded),
    ('num_impute', SimpleImputer()),
    ('ss', StandardScaler())
])
```

Model Iteration

Now that we have the preprocess pipeline staged for use, we will iterate through five different model types in order to compare and contrast how well each model performs with different evaluation metrics. Namely, we will iterate through:

- Logistic Regression (we will use this as our baseline model)
- K-Nearest Neighbors
- Decision Tree Classifier
- Random Forest
- XGBoost

In [22]:
```
def all_metrics(model, X_set, y_true):
    """
    Takes in a fitted model (or pipeline) as well as the X and y validatio
    and returns the accuracy score, recall score, F1 score, precision scor
    """
    # Predict
    y_preds = model.predict(X_set)

    # Accuracy
    accuracy = accuracy_score(y_true, y_preds)

    # Recall
    recall = recall_score(y_true, y_preds)

    # F1
    f1 = f1_score(y_true, y_preds)

    # Precision
    precision = precision_score(y_true, y_preds)

    # Confusion Matrix
    cnf_mtrx = confusion_matrix(y_true, y_preds)

    return accuracy, recall, f1, precision, cnf_mtrx
```

In [23]:
```
def model_comparison(models, X_set, y_true):
    """
    This function takes in a list of tuples in the form ('name of model',
    It also takes in the X and y validation or test data to be passed to t
    The function returns the metrics found from all_metrics in dictionary
    the fitted models.
    """

    accuracy_scores = []
    recall_scores = []
    f1_scores = []
    precision_scores = []
    confusion_matrices = []
    fitted_models = {}

    for model_tuple in models:
        model_name = model_tuple[0]
        model = model_tuple[1]

        # Create Pipeline with preprocess + current model
        pipe = Pipeline(steps=[
            ('preprocess', preprocess_pipe),
            (model_name, model)
        ])

        # Fit
        pipe.fit(X_train, y_train)
        fitted_models['{}'.format(model_name)] = pipe

        # Find Metrics
        accuracy, recall, f1, precision, cnf_mtrx = all_metrics(pipe, X_se
        accuracy_scores.append(accuracy)
        recall_scores.append(recall)
        f1_scores.append(f1)
        precision_scores.append(precision)
        confusion_matrices.append(cnf_mtrx)

    return {'Accuracy': accuracy_scores, 'Recall': recall_scores, 'F1': f1
           'Precision': precision_scores}, confusion_matrices, fitted_mode
```

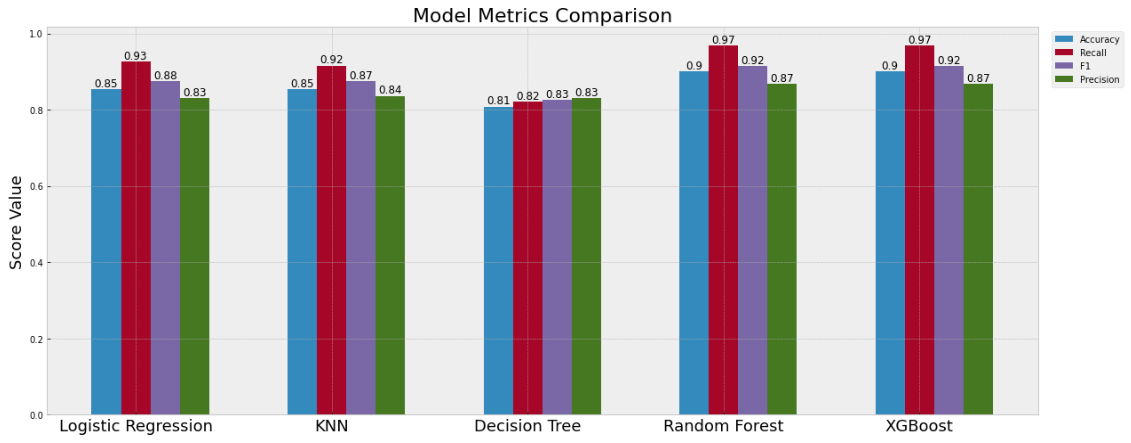
Now that we have some helper functions that can iterate through our models and determine the metrics for us, let's pass through the five models we wish to run.

In [24]:
```
models = [('logistic_regression', LogisticRegression()),
          ('knn', KNeighborsClassifier()),
          ('classification_tree', DecisionTreeClassifier(random_state=7)),
          ('random_forest', RandomForestClassifier(random_state=7)),
          ('xgboost', XGBClassifier(random_state=7)) ]

metrics, confusion_matrices, fitted_models = model_comparison(models, X_va
```

Visualize Results

In [25]:
```
def graph_metrics(model_names, metrics_list):
    """
    This function takes in a list of model names and a dictionary of lists
    model respectively.  It creates a bar chart that displays each metric
    """
    width = 0.15
    multiplier = 0
    x = np.arange(len(model_names))

    fig, ax = plt.subplots(figsize=(20,8))

    for score_name, score_data in metrics_list.items():
        offset = width * multiplier
        ax.bar(x + offset, score_data, width, label=score_name)
        for x_n, s in zip(x+offset, score_data):
            ax.text(x_n, s, round(s,2), ha='center', va='bottom', fontsize
        multiplier +=1

    ax.set_ylabel('Score Value', fontsize=18)
#     ax.set_xlabel('Model', fontsize=18)
    ax.set_title('Model Metrics Comparison', fontsize=22)
    ax.set_xticks(x + width)
    ax.set_xticklabels(model_names, fontsize=18)
    ax.legend(bbox_to_anchor=(1.05,1), loc='upper center');
```

In [26]:
```
# Visualize each model's results
models_names = ('Logistic Regression', 'KNN', 'Decision Tree', 'Random For

graph_metrics(models_names, metrics)
```

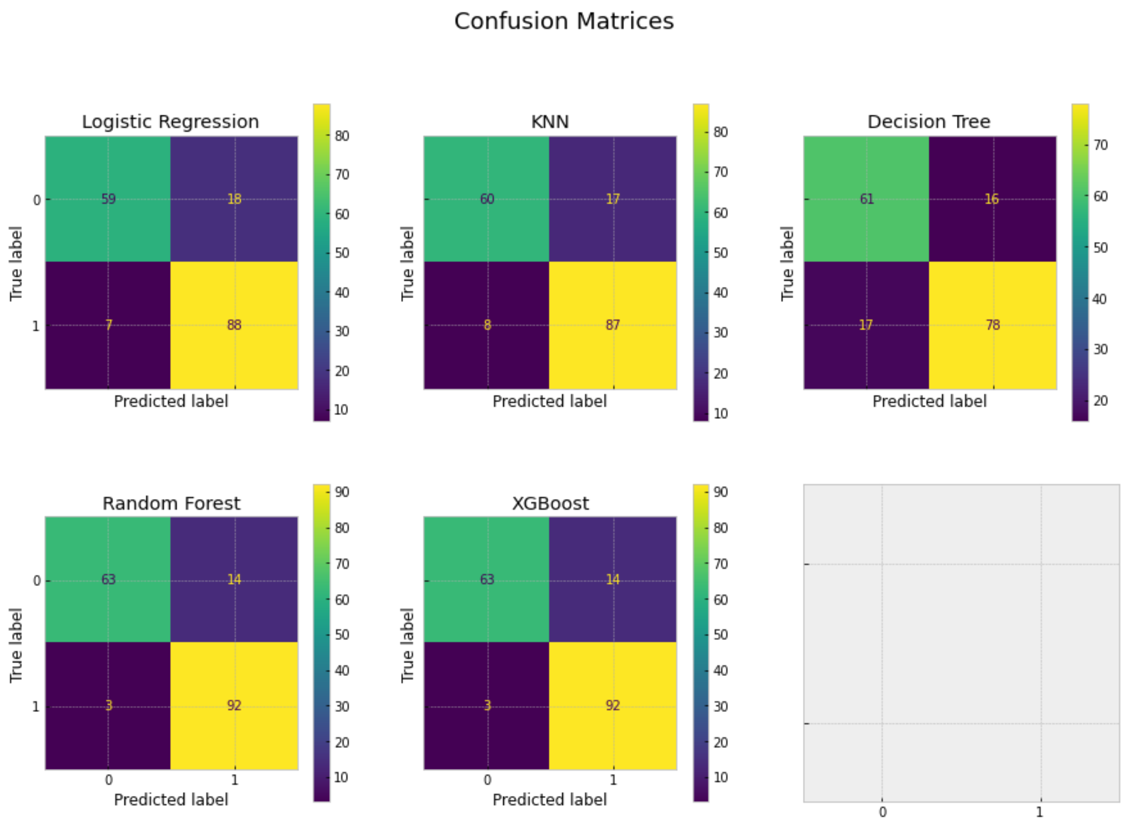


It seems that our Random Forest and XGBoost models are performing the best overall. Keeping in mind that the Logistic Regression is our 'baseline' to compare to, the K-Nearest Neighbor and Decision Tree models are both performing worse overall. This is not completely surprising as both of these models do tend to overfit without any hyperparameter tuning.

It appears that our Random Forest and XGBoost models have the exact results for all metrics! With recall at 97%, accuracy at 90%, and precision at 87%, our models are finding most of the

In [27]:
```
# Confusion Matrix for each model
def plot_cnf_matrices(cnf_matrices, model_names):
    """
    This function takes in a list of confusion matrices as well as the nam
    derived from and outputs a confusion matrix plot for each. Note that t
    confusion matrices.
    """
    fig, ax = plt.subplots(ncols=3, nrows=2, figsize=(15,10), sharey=True,

    i = 0
    for cf in cnf_matrices:
        a = i // 3
        b = i % 3

        ax[a,b].set_title("{}".format(model_names[i]))
        disp = ConfusionMatrixDisplay(cf)
        disp.plot(ax=ax[a,b])
        i += 1

    fig.suptitle("Confusion Matrices", fontsize=18)
    plt.show()
```

In [28]:
```
plot_cnf_matrices(confusion_matrices, models_names)
```



The above confusion matrices again show that the Random Forest and XGBoost models are out performing all other models.

Hyperparameter Tuning

Now that we have evaluated our models using their default parameters, we will take the time to explore tuning the best model. Seeing as the Random Forest Model and XGBoost models have identical metrics, we will go ahead and explore hyperparameter tuning with both of these models.

We will go ahead and try a GridSearchCV as well as a RandomizedSearchCV with similar distributions. Furthermore, we will try these searches with a few different scoring values in the search.

In [29]:
```
def tuned_model_comparison(model_tuples, param_grids, scoring=None, random
    """
    This model takes in a list of tuples formated ('model_name', model) fo
    It also takes in a list of parameter girds (or distributions) for each
    performs a grid search, or randomized search if 'random' is set to Tru
    best parameters, and best score for each model.
    """
    best_results = {}

    for model_tuple, grid in zip(model_tuples, param_grids):

        # Build the estimator for the search
        model_name = model_tuple[0]
        model = model_tuple[1]
        estimator = Pipeline(steps=[
            ('preprocess', preprocess_pipe),
            (model_name, model)
        ])

        # Create Grid Search object
        skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=7)
        if random == True:
            gs_pipe = RandomizedSearchCV(estimator=estimator, param_distri
        else:
            gs_pipe = GridSearchCV(estimator=estimator, param_grid=grid, s

        # Fit
        gs_pipe.fit(X_train, y_train)

        # Results
        best_results[model_name] = {'estimator': gs_pipe.best_estimator_,
                                    'params': gs_pipe.best_params_,
                                    'score': gs_pipe.best_score_}
    return best_results
```

In [30]:
```
# Create grids for GridSearchCV

forest_param_grid = {'random_forest__max_depth': [2, 8, 15, None],
                'random_forest__max_features': [1, 3, 'sqrt'],
                'random_forest__n_estimators': [10, 50, 100],
                'random_forest__min_samples_split': [30, 10, 2]}

xgb_param_grid = {'xgboost__learning_rate': [0.01, 0.1, 0.8],
                  'xgboost__max_depth': [2, 5, 15],
                  'xgboost__n_estimators': [50, 100, 200],
                  'xgboost__gamma': [0.1, 2, 5, 10]}

param_grids = [forest_param_grid, xgb_param_grid]
```

In [31]:
```
# Create distributions for RandomizedSearchCV

rand_forest_param_grid = {'random_forest__max_depth': stats.randint(2, 15)
                'random_forest__max_features': stats.uniform(0,1),
                'random_forest__n_estimators': stats.randint(10, 100),
                'random_forest__min_samples_split': stats.uniform(0.01, 0.9

rand_xgb_param_grid = {'xgboost__learning_rate': stats.uniform(0.01, 0.08)
                  'xgboost__max_depth': stats.randint(2, 15),
                  'xgboost__n_estimators': stats.randint(50,200),
                  'xgboost__gamma': stats.uniform(.1, 10)}

rand_param_grids = [rand_forest_param_grid, rand_xgb_param_grid]
```

In [32]:
```
# Models to tune
tune_models = [('random_forest', RandomForestClassifier(random_state=7)),
               ('xgboost', XGBClassifier(random_state=7)) ]
```

In [33]:
```
# Grid Search Results
best_models_std = tuned_model_comparison(tune_models, param_grids)
best_models_precision = tuned_model_comparison(tune_models, param_grids, s
best_models_recall = tuned_model_comparison(tune_models, param_grids, scor

# Randomized Search Results
best_models_rand = tuned_model_comparison(tune_models, rand_param_grids, r
best_models_rand_precision = tuned_model_comparison(tune_models, rand_para
best_models_rand_recall = tuned_model_comparison(tune_models, rand_param_g
```

Visualize Results

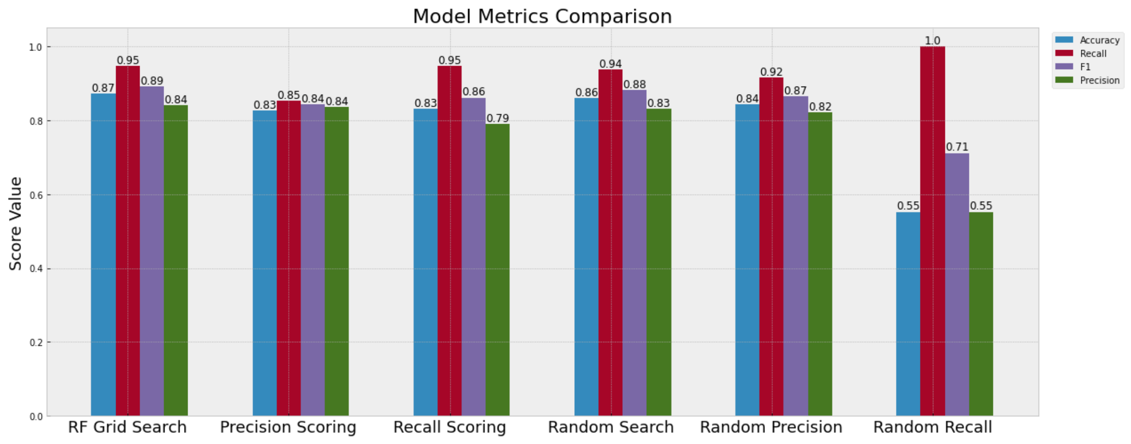
Now that we have the results from all our searches stored, we will evaluate with our helper all_metrics, graph_metrics, and plot_cnf_matrices functions.

In [34]:

```
# Condense all results into one list
searched_models = [best_models_std, best_models_precision, best_models_rec
                   best_models_rand, best_models_rand_precision, best_model

# Create lists of metrics to feed to graph_metrics function later
rf_accuracy_scores = []
rf_recall_scores = []
rf_f1_scores = []
rf_precision_scores = []
rf_confusion_matrices = []

xgb_accuracy_scores = []
xgb_recall_scores = []
xgb_f1_scores = []
xgb_precision_scores = []
xgb_confusion_matrices = []


# Iterate through results to find and store metrics for both Random Forest

for model in searched_models:

    # Find and append metrics from the random forest model
    accuracy, recall, f1, precision, cnf_mtrx = all_metrics(model['random_
                                                       X_val, y_val)
    rf_accuracy_scores.append(accuracy)
    rf_recall_scores.append(recall)
    rf_f1_scores.append(f1)
    rf_precision_scores.append(precision)
    rf_confusion_matrices.append(cnf_mtrx)

    # Find and append metrics from the xgboost model
    accuracy, recall, f1, precision, cnf_mtrx = all_metrics(model['xgboost
                                                       X_val, y_val)
    xgb_accuracy_scores.append(accuracy)
    xgb_recall_scores.append(recall)
    xgb_f1_scores.append(f1)
    xgb_precision_scores.append(precision)
    xgb_confusion_matrices.append(cnf_mtrx)

# Create dictionary of metrics for each model type
rf_metrics = {'Accuracy': rf_accuracy_scores, 'Recall': rf_recall_scores,
           'Precision': rf_precision_scores}
xgb_metrics = {'Accuracy': xgb_accuracy_scores, 'Recall': xgb_recall_score
           'Precision': xgb_precision_scores}
```

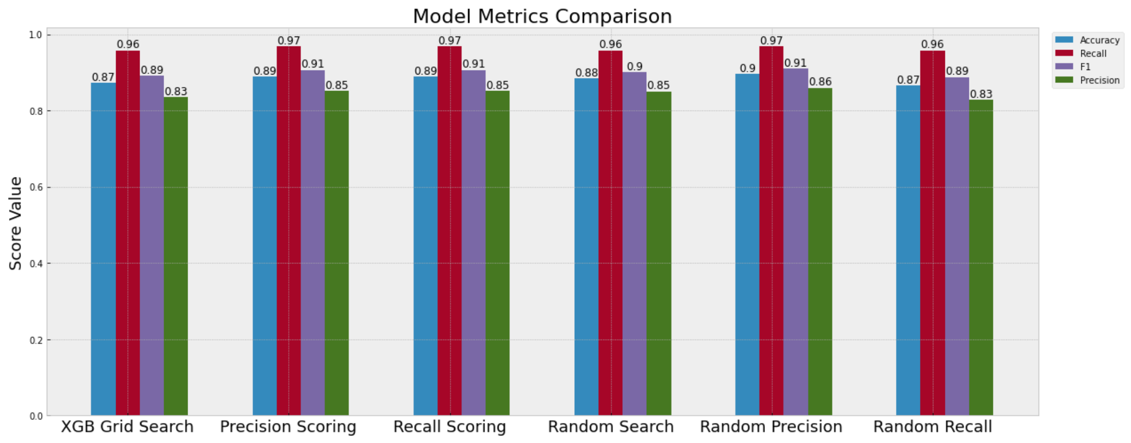
In [35]:

```
# Graph Random Forest Search Results
rf_names = ['RF Grid Search', 'Precision Scoring', 'Recall Scoring', 'Rand
            'Random Precision', 'Random Recall']
graph_metrics(rf_names, rf_metrics)
```

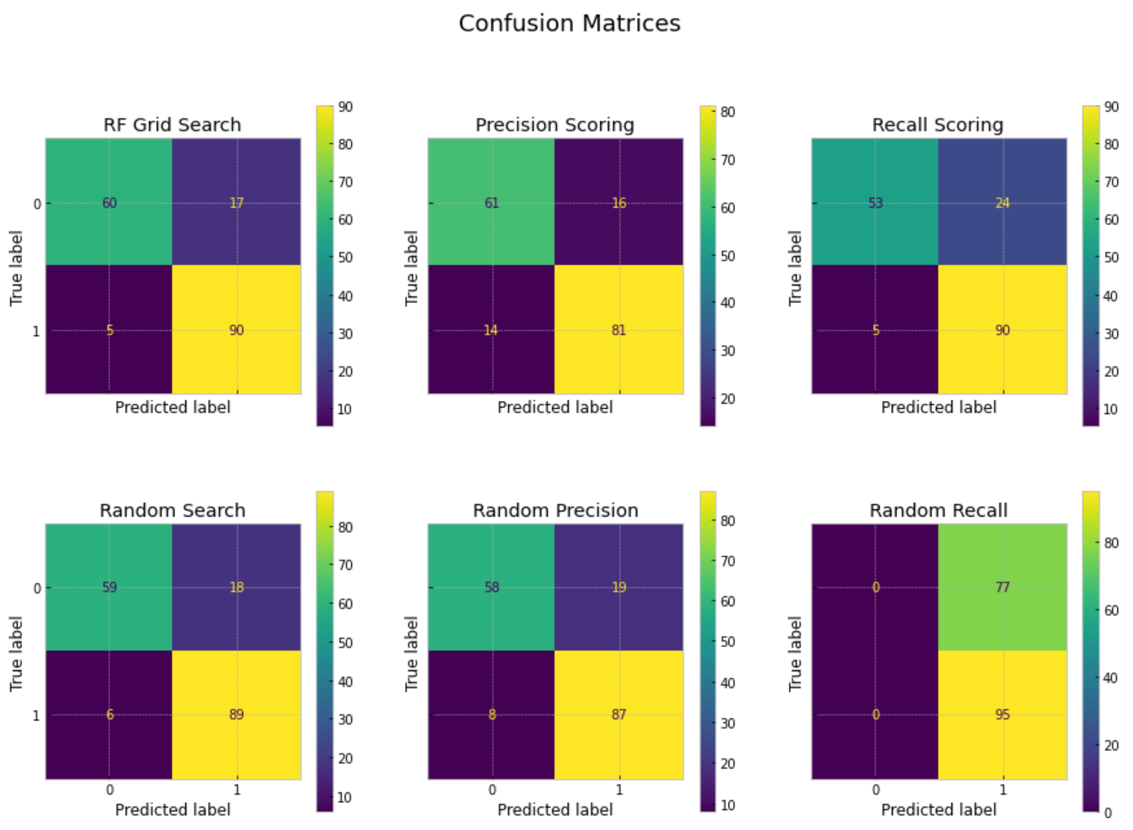


In [36]:

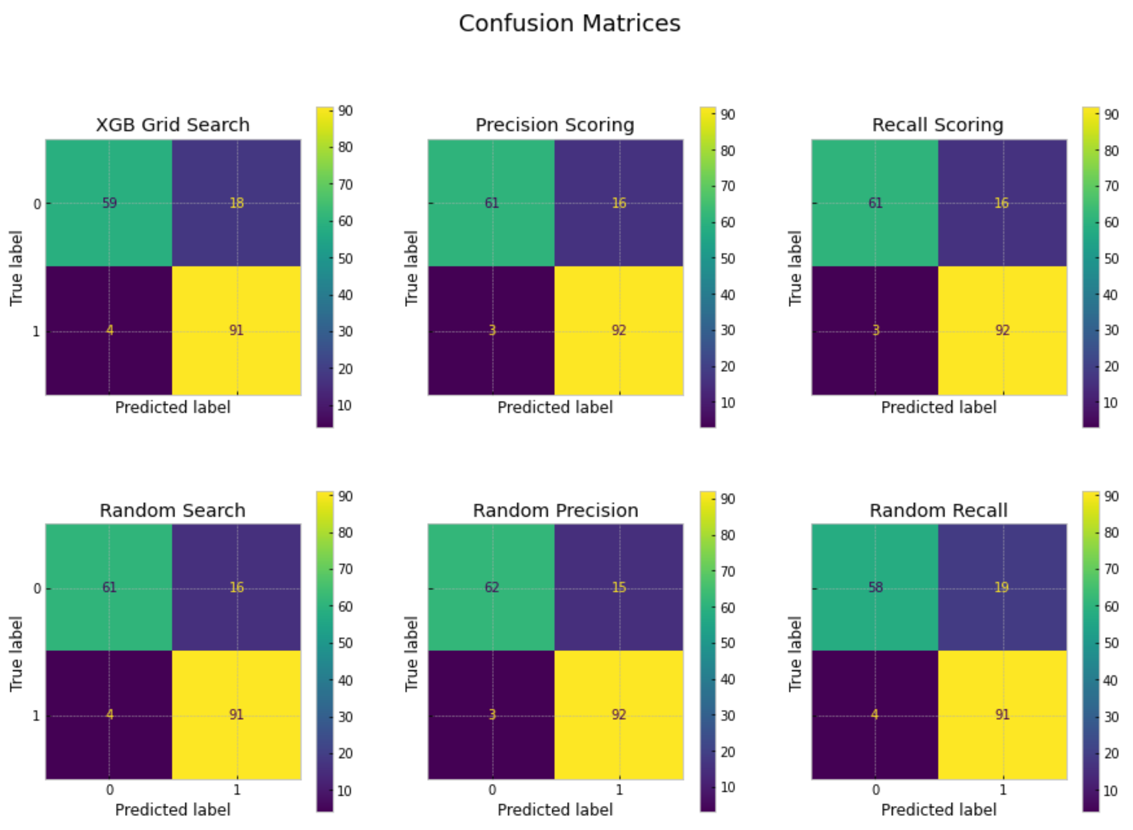
```
# Graph XGBoost Search Results
xgb_names = ['XGB Grid Search', 'Precision Scoring', 'Recall Scoring', 'Ra
             'Random Precision', 'Random Recall']
graph_metrics(xgb_names, xgb_metrics)
```



```
In [37]:  # Confusion Matrices for Random Forest Search Results
          plot_cnf_matrices(rf_confusion_matrices, rf_names)
```



```
In [38]:  # Confusion Matrices for XGBoost Search Results
          plot_cnf_matrices(xgb_confusion_matrices, xgb_names)
```



Seeing as all those visuals can be a bit overwhelming, let's pull out the data from the best models in order to compare and contrast only these. The following out performed all other models:

- Random Forest model with default hyperparameters
- XGBoost model with default hyperparameters
  - Since the metrics were identical for these two, they will be represented as 'default_params' in the DataFrame below
- XGBoost using a GridSearchCV, with scoring set to Recall, to choose hyperparameters
- Random Forest with a RandomizedSearchCV to find hyperparameters (default scoring)
- XGBoost with a RandomizedSearchCV to find hyperparameters (default scoring)
- XGBoost with a RandomizedSearchCV, with scoring set to Precision, to find hyperparameters

In [39]:

```
pd.DataFrame({'default_params': [90,97,92,87], 'xgb_recall': [89, 97, 91,
               'xgb_random': [90, 96, 91, 88], 'xgb_random_precision':[88,
              index=['Accuracy', 'Recall', 'F1', 'Precision'])
```

Out[39]:

	default_params	xgb_recall	rf_random	xgb_random	xgb_random_precision
Accuracy	90	89	90	90	88
Recall	97	97	97	96	97
F1	92	91	91	91	90
Precision	87	85	86	88	84

We are still seeing that the two best models are the Random Forest and XGBoost with the default parameters, however the Random Forest model that was found utilizing the RandomizedSearchCV (with default scoring) comes at a close second. Let's take a look at the classification report for these three models in order to get a better idea of how they are performing in relation to each target class.

In [40]:

```
# Classification Report for the XGBoost model with default hyperparameters

preds = fitted_models['xgboost'].predict(X_val)
print(classification_report(y_val, preds))
```

```
              precision    recall  f1-score   support

           0       0.95      0.82      0.88        77
           1       0.87      0.97      0.92        95

    accuracy                           0.90       172
   macro avg       0.91      0.89      0.90       172
weighted avg       0.91      0.90      0.90       172
```

In [41]:

```
# Classification Report for the Random Forest model with default hyperpara

preds = fitted_models['random_forest'].predict(X_val)
print(classification_report(y_val, preds))
```

```
              precision    recall  f1-score   support

           0       0.95      0.82      0.88        77
           1       0.87      0.97      0.92        95

    accuracy                           0.90       172
   macro avg       0.91      0.89      0.90       172
weighted avg       0.91      0.90      0.90       172
```

In [42]:

```
# Classification Report for the Random Forest Model with a RandomizedSearc
    # hyperparameters (default scoring)

preds = best_models_rand['random_forest']['estimator'].predict(X_val)
print(classification_report(y_val, preds))
```

```
              precision    recall  f1-score   support

           0       0.91      0.77      0.83        77
           1       0.83      0.94      0.88        95

    accuracy                           0.86       172
   macro avg       0.87      0.85      0.86       172
weighted avg       0.87      0.86      0.86       172
```

From all of the above, we can conclude that the best model available to us is either the Random Forest or XGBoost model with default hyperparameters.

The Random Forest model found with the RandomizedSearchCV is simply not quite as accurate. Out of our two most performant models, we will move forward with the Random Forest. Both would be applicable in this senario to move forward.

Final Model

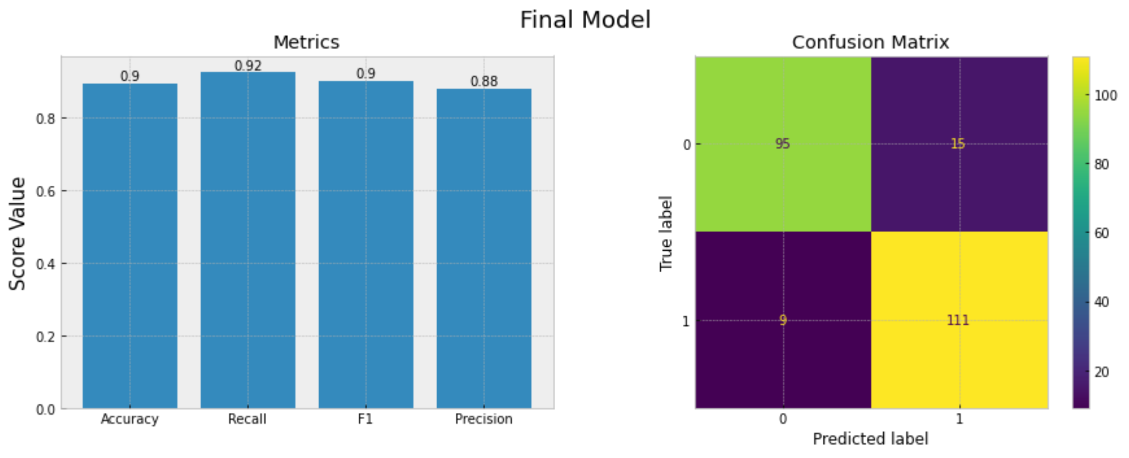
Now that we have chosen our final model, we will train it with all of our training data (not just the 80% we have been using in order to have a validation set for the model iteration process), and evaluate it with the completely unseen test data.

In [43]:
```
final_model = Pipeline(steps=[
    ('preprocess', preprocess_pipe),
    ('random_forest', RandomForestClassifier(random_state=7))
])
final_model.fit(X_train_all, y_train_all)
final_accuracy, final_recall, final_f1, final_precision, final_cnf_mtrx =
final_metrics = [final_accuracy, final_recall, final_f1, final_precision]
```

Evaluation

Let's see how well our final model performs on the test data.

In [44]:
```
metric_names = ['Accuracy', 'Recall', 'F1', 'Precision']

fig, ax = plt.subplots(ncols=2, figsize=(15,5))

ax[0].bar(metric_names, final_metrics)
for name, score in zip(metric_names,final_metrics):
    ax[0].text(name, score, round(score,2), ha='center', va='bottom')
ax[0].set_title('Metrics')
ax[0].set_ylabel('Score Value', fontsize=15)

disp = ConfusionMatrixDisplay(final_cnf_mtrx)
disp.plot(ax=ax[1])
ax[1].set_title('Confusion Matrix')

fig.suptitle('Final Model', fontsize=18);
```



Furthermore, let's look at how important each feature was in our Random Forest.

In [45]:

```
# Feature Names for Graph
feature_names = list(data.columns)[:-1]

# Run Permutation Importance
perm_results = permutation_importance(
    final_model, X_test, y_test, n_repeats=50,
    random_state=7, n_jobs=2)

# Format results for graph
forest_importances = pd.Series(perm_results.importances_mean, index=featur
```

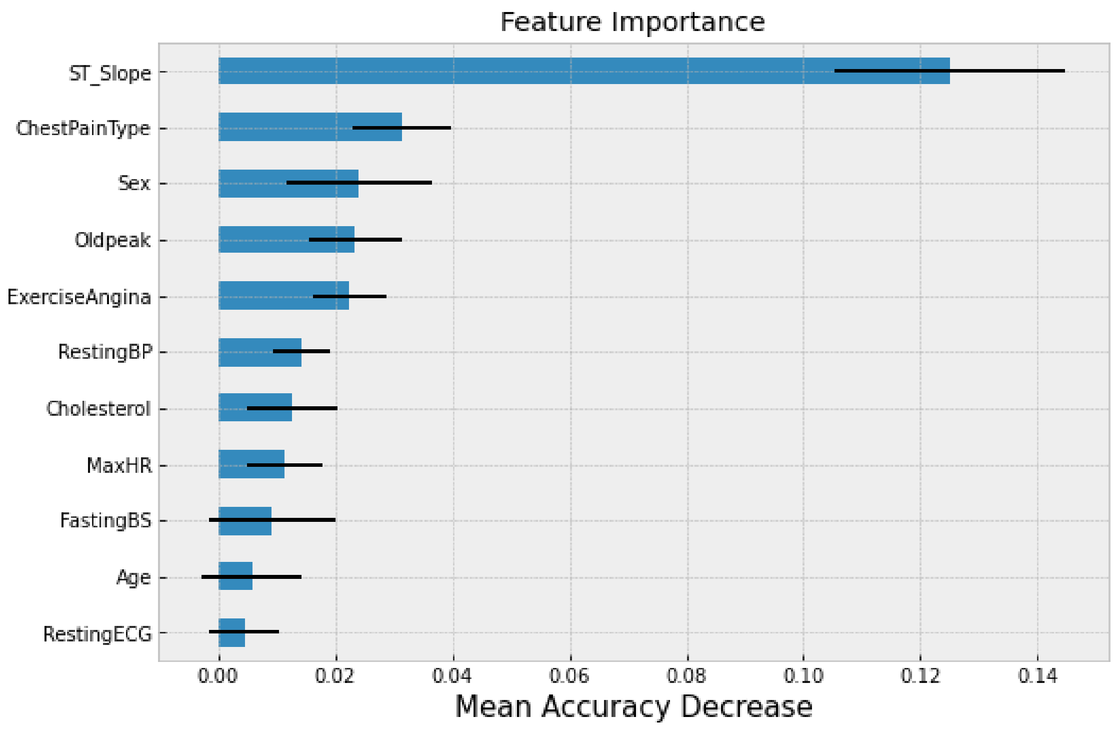
In [46]:

```
# Print Classification Report
preds = final_model.predict(X_test)
print(classification_report(y_test, preds))
```

```
              precision    recall  f1-score   support

           0       0.91      0.86      0.89       110
           1       0.88      0.93      0.90       120

    accuracy                           0.90       230
   macro avg       0.90      0.89      0.90       230
weighted avg       0.90      0.90      0.90       230
```

In [47]:

```python
# Graph Feature Importances
fig, ax = plt.subplots(figsize=(9,6))

forest_importances.sort_values().plot.barh(
    xerr=perm_results.importances_std, ax=ax)

ax.set_title('Feature Importance')
ax.set_xlabel('Mean Accuracy Decrease', fontsize=15);
```



From the above metrics graphs we can see that our final model is 90% accurate, has an overall recall score of 92%, and a recall score of 93% for the target class 1 (having heart disease). These scores are less than what we were seeing with the validation data, but are still quite good results. From the confusion matrix, we can see that of the 230 data points in the test set, 111 were correctly identified as having heart disease by the model. The model missed 9 data points that did indeed have heart disease (misclassified as healthy patients), and misclassified only 15 healthy data points as having heart disease. Keeping in mind that the health care provider's main concern was to capture as many patients as possible without too many healthy patients classified as being ill, the model is performing quite well.

The feature importance graph above gives some insight to the model itself. We can see that the shape of the ST segment during exercise is by far the most important variable; the model's accuracy decreased by about 12% without this feature. The second most important variable for the model is the Chest Pain Type -- this is a rather intuitive result as having chest pain generally means something in the heart is awry.

It is interesting to note that the two least important features for the model are RestingECG and Age. With the ST segment shape during exercise being the most important, one would think that resting ECG would be relevant as well. However, in this model, the importance of the resting ECG is widely variable (with the mean accuracy actually increasing for a few of the trees in the forest when this feature is not included). Secondly, one would generally assume that age would be a more important indicator of heart disease, but again we see a large variation in how important the age feature is in our forest, with a few estimators performing better without this feature

Recommendations

First and foremost, this model is a tool to assist the health care provider in identifying patients with heart disease prior to a crisis. The providers can utilize the model by inputting patient information (ideally already sitting in their database) and advising patients for follow up testing if the model predicts that they have heart disease. Preventative treatment can be implemented which is not only most cost effective for the patient, but also can save lives.

Secondly, seeing as the shape of the ST segment during exercise is the most important feature to this model, it is recommended that the health care providers include screening for this feature during patient checkups. Currently this is not a standard measurement taken in normal doctor's visits. However, if a patient's ECG during exercise is more regularly recorded, then the model can be put to better use, and early diagnosis is more likely.

Lastly, seeing as chest pain type is the second most important feature in the model, it is recommended that the health care providers investigate ways to acquire further information about patients' chest pain. This could be done in a cheaper and more widespread manner than acquiring ECGs during exercise for patients. For example, a survey could be sent out to all patients concerning any chest pain, and those with qualifying responses could be recommended to come in for further testing.