Homework #3

Carl Xi

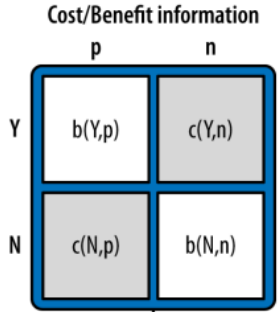
(put your name above (incl. any nicknames) 5pt)

Total grade: _____ out of ___150___ points

1) **(20 points) Imagine that you work for an online advertising company that has just been hired to advertise a new local restaurant online. Let's say that it costs $0.015 to present a coupon ad to online consumers. If a consumer cashes in your coupon, you stand to earn $5.**

a) Given this information, what would your cost/benefit matrix be? Explain your reasoning briefly.

[2.5 points for each correct cell in cost/benefit matrix]



Since there is no 'cost' a consumer cashing in our coupon, our net earning is purely $5. Our cost is $0.015 for advertisement. In this case, our cost/benefit matrix looks something like the matrix shown on the left. b(Y,p) is consumers who saw the ad and cashed in the coupon. c(Y,n) is consumers who saw the ad and did not cash in the coupon. c(N,p) is consumers who did not see the ad but would have cashed in the coupon if they saw the ad. b(N,n) is consumers who did not see the ad and would not cash in the coupon even if they did. This way, we can calculate

Cost/Benefit Matrix	P	N
Y	V= $5 - $0.015 = $4.985	V = $0 - $0.015 = $ -0.015
N	V= $0	V = $0

b) Suppose that I build a classifier that provides the following confusion matrix. What is the expected value of that classifier? Justify your answer.

[3 points for expected value formula, 7 points for correct application of expected value formula]

	Positive	Negative
Positive	560	70
Negative	120	450

We know from class that expected value is calculated using the formula below:

- The general form of an expected value calculation:

$$EV = p(o_1)\times v(o_1) + p(o_2)\times v(o_2) + p(o_3)\times v(o_3) + ..$$

e.g., for binary classification $EV = p(o_1)\times v(o_1) + (1 - p(o_1))\times v(o_2)$

Adding up the total, we get 1200.

The Confusion Matrix above then can be transformed into a table of $p(o_i)$ values:

	Positive	Negative
Positive	560/1200 = 7/15 = 0.467	70/1200 = 7/120 = 0.0583
Negative	120/1200 = 1/10 = 0.1	450/1200 = 3/8 = 0.375

Multiplying our $v(o_i)$ values with our $p(o_i)$ values cell wise gives us the following, which is our Expected Value:

$$EV = \left(\frac{560}{1200} * 4.985\right) + \left(\frac{120}{1200} * 0\right) + \left(\frac{70}{1200} * -0.015\right) + \left(\frac{450}{1200} * 0\right) = 2.325458 \text{ or } \underline{\mathbf{\$2.33}}$$

The expected value computation provides a framework that is useful in organizing thinking about data-analytic problems • It decomposes data-analytic thinking into • the structure of the problem, • the elements of the analysis that can be extracted from the data, and • the elements of the analysis that need to be acquired from other sources

2) (25 points) You have a fraud detection task (predicting whether a given credit card transaction is "fraud" vs. "non-fraud") and you built a classification model for this purpose. For any credit card transaction, your model estimates the probability that this transaction is "fraud". The following table represents the probabilities that your model estimated for the validation dataset containing 10 records.

Actual Class (from validation data)	Estimated Probability of Record Belonging to Class "fraud"
fraud	0.95
fraud	0.91
fraud	0.75
non-fraud	0.67
fraud	0.61
non-fraud	0.46
fraud	0.42
non-fraud	0.25
non-fraud	0.09
non-fraud	0.04

Draw an ROC curve for your model (use at least six different thresholds to draw the ROC curve).
[10 points showing your calculations for each threshold ->i.e., the corresponding metrics you need for ROC curve, 5 points for correctly labeling axes in ROC graph, 10 points for correctly depicting the ROC graph – you can do this by hand or some using software such as Excel or Python]

By drawing a stepwise chart that depicts dropping cutoff, we can easily visualize the total number of TP, TN, FP, FNs at each step:

Actual Class	Estimated Probability of "Fraud"	Cutoff = 0.99	Cutoff = 0.95	Cutoff = 0.91	Cutoff = 0.75	Cutoff = 0.67	Cutoff = 0.61	Cutoff = 0.46	Cutoff = 0.42	Cutoff = 0.25	Cutoff = 0.09	Cutoff = 0.04
fraud	0.95	FN	TP	TP	TP	TP	TP	TP	TP	TP	TP	TP
fraud	0.91	FN	FN	TP	TP	TP	TP	TP	TP	TP	TP	TP
fraud	0.75	FN	FN	FN	TP	TP	TP	TP	TP	TP	TP	TP
non-fraud	0.67	TN	TN	TN	TN	FP	FP	FP	FP	FP	FP	FP
fraud	0.61	FN	FN	FN	FN	FN	TP	TP	TP	TP	TP	TP
non-fraud	0.46	TN	TN	TN	TN	TN	TN	FP	FP	FP	FP	FP
fraud	0.42	FN	FN	FN	FN	FN	FN	FN	TP	TP	TP	TP
non-fraud	0.25	TN	TN	TN	TN	TN	TN	TN	TN	FP	FP	FP
non-fraud	0.09	TN	TN	TN	TN	TN	TN	TN	TN	TN	FP	FP
non-fraud	0.04	TN	TN	TN	TN	TN	TN	TN	TN	TN	TN	FP

***The bolded black line depicts the cutoff**

The ROC Graph/Curve plots classifier's performances with FP_{rate} on the x-axis and the TP_{rate} on the y-axis:

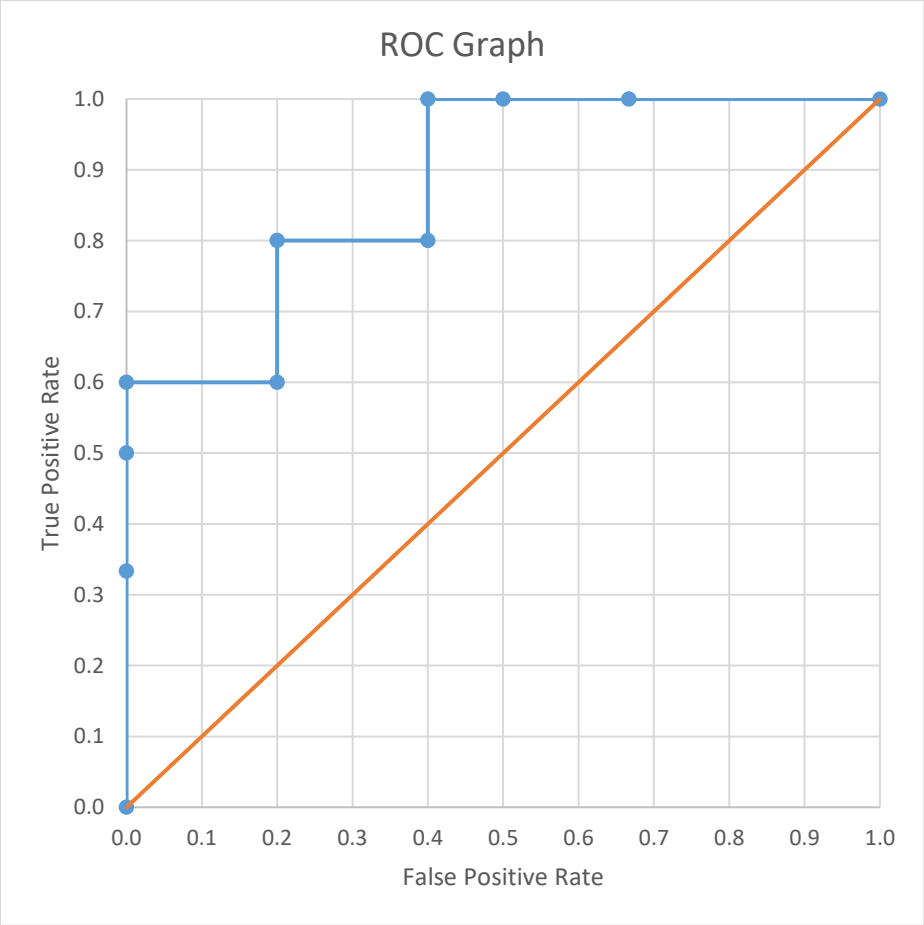
$$FP_{rate} = \frac{FP}{N} = \frac{FP}{FP+TN}$$ ← Fraction of negative examples incorrectly classified

$$TP_{rate} = \frac{TP}{P} = \frac{TP}{TP+FN}$$ ← Fraction of positive examples correctly classified (recall)

Using the FP_{rate} and TP_{rate} formulas below, we can easily calculate the FP and TP rates for each cutoff:

Cutoff	Sum (TP)	Sum (TN)	Sum (FP)	Sum (FN)	FP Rate	TP Rate
Cutoff > 0.95	0	5	0	2	0.0	0.0
0.95 > Cutoff > 0.91	1	5	0	2	0.0	0.3
0.91 > Cutoff > 0.75	2	5	0	2	0.0	0.5
0.75 > Cutoff > 0.67	3	5	0	2	0.0	0.6
0.67 > Cutoff > 0.61	3	4	1	2	0.2	0.6
0.61 > Cutoff > 0.46	4	4	1	1	0.2	0.8
0.46 > Cutoff > 0.42	4	3	2	1	0.4	0.8
0.42 > Cutoff > 0.25	5	3	2	0	0.4	1.0
0.25 > Cutoff > 0.09	5	2	2	0	0.5	1.0
0.09 > Cutoff > 0.04	5	1	2	0	0.7	1.0
Cutoff < 0.04	5	0	2	0	1.0	1.0

With the FP_{rate} and TP_{rate} calculated, we can now plot our ROC curve. The diagonal red line is not part of the ROC curve, as it just used to visualize the boundary under which a classifier's performance would be even worse than random guessing. The (0,0) point represents a cutoff where the classifier never issues a positive classification (no cost but not benefit as well), while the (1,1) point represents a cutoff where the classifier unconditionally issues positive classifications. The graph is depicted below:



Logistic regression, knn or decision tree are all possible

3) (100 points) [Mining publicly available data] Use Python for this Exercise.
Please use the dataset on breast cancer research from this link: http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data We have worked with this dataset in HW2. The description of the data and attributes can be found at this link: http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.names . Each record of the data set represents a different case of breast cancer. Each case is described with 30 real-valued attributes: attribute 1 represents case id, attributes 3-32 represent various physiological characteristics, and attribute 2 represents the type (benign or malignant). If the dataset has records with missing values, you can filter out these records using Python. Alternatively, if the data set has missing values, you could infer the missing values.

[We have seen this data before – No need to explore the data for this exercise]

a) **We would like to perform a predictive modeling analysis on this same dataset using the a) decision tree, b) the k-NN technique and c) the logistic regression technique. Using the nested cross-validation technique, try to optimize the parameters of your classifiers in order to improve the performance of your classifiers (i.e., f1-score) as much as possible. Please make sure to always use a random state of "42" whenever applicable. What are your optimal parameters and what is the corresponding performance of these classifiers? Please provide screenshots of your code and explain the process you have followed.**

   [part a is worth 25 points in total:
   7 points for correctly optimizing at least **two** parameters for the Decision Tree and providing **screenshots/explaining what you are doing** and the corresponding results
   7 points for correctly optimizing at least two parameters for the kNN and providing screenshots/explaining what you are doing and the corresponding results
   7 points for correctly optimizing at least two parameters for the Logistic Regression and providing screenshots/explaining what you are doing and the corresponding results
   4 points for contrasting their performance of all three algorithms and discussing which one would you prefer to use] – justify why it is the case

Points will not be deducted for not getting the highest possible results, but have to become very familiar with this

As always, we start our process with the import of all necessary packages and setting our root directory to the relevant folder. It is important to note that we have set the random seed to 42, and you will continue to see a lot of "set seed to 42" calls throughout the notebook. You can see this in the screenshot on the next page.

Data Import & Prep

In [1]:
```
# To write a Python 2/3 compatible codebase, the first step is to add this line to the top of each module
from __future__ import division, print_function, unicode_literals
from IPython.display import Image
from sklearn import linear_model, neighbors, datasets, metrics, tree        # The sklearn.linear_model module implements general
from sklearn.linear_model import LogisticRegression # Logistic regression classifier class
from matplotlib.colors import ListedColormap # Learn more about matplotlib.colors here https://matplotlib.org/3.1.1/api/colors_a
import matplotlib.pyplot as plt #pyplot is matplotlib's plotting framework https://matplotlib.org/users/pyplot_tutorial.html
import matplotlib
import scipy as sp # sp is an alias pointing to scipy
import numpy as np
import pandas as pd # pd is an alias point to pandas
import warnings
warnings.filterwarnings("ignore")
from sklearn.model_selection import train_test_split, cross_val_score, validation_curve, GridSearchCV, KFold, StratifiedKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder
# StandardScaler Standardize features by removing the mean and scaling to unit variance
# LabelEncoder Encode labels with value between 0 and n_classes-1
# Cross_val_score Evaluate a score by cross-validation

# The sklearn.metrics module includes score functions, performance metrics and pairwise metrics
# and distance computations.
# https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics
#----------
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, f1_score, cohen_kappa_score,  roc_curve, au
from sklearn.tree import export_graphviz, DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
# If you don't have graphviz package, you need to install it https://anaconda.org/anaconda/graphviz
# How to install Graphviz with Anaconda 1
# conda install -c anaconda graphviz
!pip install graphviz
import itertools
import graphviz
import os

# Seed the generator to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline

# Dynamically change the default rc settings in a python script
# See documentation for a complete list of parameters https://matplotlib.org/users/customizing.html
plt.rcParams['axes.labelsize'] = 14  # fontsize of the x any y labels
plt.rcParams['xtick.labelsize'] = 12 # fontsize of the tick labels
plt.rcParams['ytick.labelsize'] = 12 # fontsize of the tick labels
#-----------------
# Matplotlib inline allows the output of plotting commands will be displayed inline
%matplotlib inline

# Root Directory
PROJECT_ROOT_DIR = "C:\\Users\\carlj\\OneDrive\\Documents\\School-MSBA\\Classes\\Fall\\Intro. to Business Analytics\\HW3"
```

```
Requirement already satisfied: graphviz in c:\programdata\anaconda3\lib\site-packages (0.13)
```

After which, we imported our downloaded dataset, made sure the headers made sense, isolated the dependent variable, recoded Malignant and Benign to 1 and 0 respectively, and split the dataset into 70% train and 30% test using stratified shuffling. We also created standardized versions of training and test datasets using the metrics derived from the training dataset (e.g. standard deviation and mean) for kNN problems later down in the document.

Professor Todri explicitly stated to skip data exploration this week as the dataset is the same as last week, so we will jump straight into parameter tuning. We start by declaring inner and outer cross validation folds. We kept all cross validation folding to 5 folds with shuffling and random state set to 42 to ensure consistency. For all my optimization calls, I kept the parameters mostly intact from the examples as the parameters chosen were already the best ones to tune. I did also search online for other parameters and play around with them, but they all seemed to have minimal effect on the results.

Professor Todri clarified in class that we will be using the f1 score to evaluate model performance throughout this entire homework, so please expect to see that being the case in this optimization run and also future runs and charts in parts b, c, and d, of this question.

We know that the f1 score (also known as the f-measure) is the harmonic mean between precision and recall and ranges between 0 and 1, with 0 representing worst performance and 1 representing best performance. As well, it takes into

account both false positive and false negatives and is better at dealing with uneven class distribution (which is the case with our dataset). Lastly, the cost of false positives and false negatives in our scenario drastically varies as it deals with potential life and death situations, and accuracy only works well for scenarios where the cost of false positives and false negatives are roughly equal. Thus, f1 is by far the best metric to evaluate our 3 models.

In [2]:
```
######################################### Imports ##########################################
hw2data = pd.read_csv("wdbc.data", header=None, names=["id","diagnosis","radius_mean","radius_stderror","radius_worst","texture_
#hw2data.rename(columns={0:"id",1:"diagnosis"},inplace=True)

###################################### Isolating the Target Variable ######################################
# Retrieving Attributes
X = hw2data.iloc[:,2:].values
# Retriving Target Variable
y = hw2data.iloc[:,1].values

le = LabelEncoder()
#print(y)               #See label before transformation
y = le.fit_transform(y) #Labels 'M' as 1 and 'B' as 0
#print(y)               #See label after transformation
#print(le.classes_)      #Show the classes that have been encoded

################################## Split the Data into 70% training and 30% test ###################################
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y)

################### Standardize Training and Testing X using metrics from Training X ############################
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Parameter Tuning

In [3]:
```
################################### Parameter Tuning ###################################
# Exhaustive search over specified parameter values for an estimator.
# GridSearchCV implements a “fit” and a “score” method.
# It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform”
# if they are implemented in the estimator used.
# The parameters of the estimator used to apply these methods are optimized by cross-validated
# grid-search over a parameter grid.

inner_cv = KFold(n_splits=5, shuffle=True, random_state=42)#Interestingly both calls are the same but yield different results
outer_cv = KFold(n_splits=5, shuffle=True, random_state=42)#need to state random state
```

As a refresh on the parameters for decision tree classifier, max_depth controls the max depth of the tree, criterion evaluates whether to use gini for Gini impurity or entropy for information gain, min_samples_leaf controls the minimum number of samples required to be a leaf node, and min_sample_split controls the minimum number of samples required to split an internal node. The optimized parameters for decision tree after our first parameter tuning are a max depth of 5, criterion of gini, max_samples_leaf of 5 and minimal sample split of 2. I have tried extending the possible range of the parameters up to 10 (e.g. min_samples_split tested for 2-10) but the results stayed the same. As well, the prompt asks for a minimum of 2 parameters to tune, which we have clearly surpassed.

We will first initialize GridSearchCV to train and tune the 3 data mining algorithms, with param_grid as the list of parameters we want to tune. After we use the training data to perform grid search, we get the best performing model’s score via best_score (f1 in this case) and look at the parameters that enabled it. Lastly, we will use the test dataset that has been independent this whole time to estimate the performance of the best_selected model, which is coded as the best_estimator attribute of the *GridSearchCV* Object and print out the results.

Decision Tree Parameter Tuning

In [4]:
```
############################## Decision Tree Parameter Tuning ##############################

# See all the parameters you can optimize here http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassif
# Choosing optimal depth of the tree
# Choosing optimal depth of the tree AND optimal splitting criterion
# Choosing depth of the tree AND splitting criterion AND min_samples_leaf AND min_samples_split
gs_dt = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42),
                  param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None], 'criterion':['gini','entropy'],
                               'min_samples_leaf':[1,2,3,4,5],
                               'min_samples_split':[2,3,4,5]}],
                  scoring='f1', # Specifying multiple metrics for evaluation
                  cv=inner_cv,
                  n_jobs=4)

gs_dt = gs_dt.fit(X,y)
print("\nDecision Tree Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_dt.best_score_)
print("Optimal Parameter: ", gs_dt.best_params_) # Parameter setting that gave the best results on the hold out data.
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest
nested_score_gs_dt = cross_val_score(gs_dt, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
Decision Tree Parameter Tuning
Non-nested CV F1-Score:  0.934657984394364
Optimal Parameter:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 5, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')
Nested CV F1-Score:  0.9265708104408736  +/-  0.014452515617888772
```

After optimization, the best nested cross-validation F1-score for our decision tree model was 0.927 +/- 0.014. This is a pretty high considering the range of 0-1, but we will have to wait to see how it performs against the other two optimized models, especially considering the cost of false negatives in our scenario (not detecting cancer).

The optimization process for our Logistic Regression model is very similar. I decided to stick with adjusting C and penalty. C, being lambda/1, controls how much to penalize overfitting. As a smaller lambda equals viewing penalty as less important, we ideally want a smaller C but also high model performance (f1 score this case). The penalty controls whether we use l1 or l2 norm penalty, which controls lasso and ridge regression respectively. Our optimization call yielded a C of 100 the use of l1 for penalty.

Logistic Regression Parameter Tuning

In [5]:
```
############################## Logistic Regression Parameter Tuning ##############################
# Choosing C parameter (i.e., regularization parameter) for Logistic Regression
# Choosing C parameter for Logistic Regression AND type of penalty (ie., l1 vs l2)
# See other parameters here http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
gs_lr = GridSearchCV(estimator=LogisticRegression(random_state=42),
                  param_grid=[{'C': [ 0.00001, 0.0001, 0.001, 0.01, 0.1 ,1 ,10 ,100, 1000, 10000, 100000, 1000000, 10000000],
                               'penalty':['l1','l2']}],
                  scoring='f1',
                  cv=inner_cv)

gs_lr = gs_lr.fit(X,y)
print("\n Logistic Regression Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_lr.best_score_)
print("Optimal Parameter: ", gs_lr.best_params_)
print("Optimal Estimator: ", gs_lr.best_estimator_)
nested_score_gs_lr = cross_val_score(gs_lr, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score:",nested_score_gs_lr.mean(), " +/- ", nested_score_gs_lr.std())
```

```
 Logistic Regression Parameter Tuning
Non-nested CV F1-Score:  0.9600639979772475
Optimal Parameter:  {'C': 100, 'penalty': 'l1'}
Optimal Estimator:  LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l1', random_state=42, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
Nested CV F1-Score: 0.9444853129388919  +/-  0.028926982011796845
```

After optimization, the best nested cross-validation F1-score for our logistic regression model was 0.944 +/- 0.029. This is a pretty high considering the range of 0-1, and outperforms our decision tree model but, but has a higher +/- deviation and we need to see how the kNN model performs first before picking the ‘best’ model.

The last model to be tuned is the kNN model. I decided to play around with the number of neighbors (number of neighbors to use), weights (how neighbors of different distance are weighted) and p (power parameter for the minkowski metric, with 1=manhattan_distance, 2= Euclidean_distance and any other arbitrary p the minkowski_distance of (l_p) is used). Our optimization call yielded a p of 1 (manhattan_distance), uniform weighting, and 5 neighbors.

kNN Parameter Tuning

In [26]:
```
################################## kNN Parameter Tuning ###################################

#We Tune our parameters using the entire standardized X set
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

# Choosing k for kNN
# Choosing k for kNN AND type of distance
gs_knn = GridSearchCV(estimator=neighbors.KNeighborsClassifier(
                            metric='minkowski'),
                  param_grid=[{'n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
                               'weights':['uniform','distance'],'p':[1,2,3,4,5,6,7,8,9,10]}],
                  scoring='f1',
                  cv=inner_cv,
                  n_jobs=4)

#print(len(y))
gs_knn = gs_knn.fit(X,y)  #X_std
print("\n kNN Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_knn.best_score_)
print("Optimal Parameter: ", gs_knn.best_params_)
print("Optimal Estimator: ", gs_knn.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highes
nested_score_gs_knn = cross_val_score(gs_knn, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
```

```
 kNN Parameter Tuning
Non-nested CV F1-Score:  0.9207223798751314
Optimal Parameter:  {'n_neighbors': 5, 'p': 1, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=1,
           weights='uniform')
Nested CV F1-Score:  0.9138948051874017  +/-  0.04025188646924722
```

After optimization, the best nested cross-validation F1-score for our k-NN regression model was 0.914 +/- 0.040. This is a pretty high considering the range of 0-1, but underperforms both the decision tree and logistic regression models. As well, this model has the highest deviation amongst all 3 models at +/- 0.04.

Amongst the 3 models, the k-nn model is the worst-performing one. With a potential f1 score of as low as 0.873 and as high as 0.953, it is surpassed by the decision tree and logistic regression models, which can perform between 0.913-0.941 and 0.915-0.973 respectively. The nested cross-validation f1 score of our logistic regression model was the best, and the one I would definitely choose/prefer to use, as it has both the highest F1 score and the best performance floor and ceiling (0.915 and 0.973 respectively). In a scenario where false positive or false negative errors can destroy families and take lives, I would want the highest F1 score that indicates the least number of error predictions.

b) Build and visualize a learning curve for the <u>logistic regression</u> technique (visualize the performance for both training and test data in the same plot). Please provide screenshots of your code and explain the process you have followed.

[part b is worth 25 points in total:
8 points for correct visualization of learning curve for **in-sample sample performance** – **show the performance for 10 different sizes** (10%,20%,30%) - provide screenshots of your code and explain the process you have followed.
8 points for correct visualization of learning curve for **out-sample sample performance** – show the performance for 10 different sizes - provide screenshots of your code and explain the process you have followed.
9 points for discussing what we can learn from this specific learning curve – what are the insights that can be drawn]

After getting the best parameters for the three models, the graphing part becomes much easier. For the Learning curve, we first define the function that plots the learning curve. We made sure that we show the performance for **<u>10</u>** different sizes, which is taken care of using the train_sizes=np.linspace(0.1,1.0,10), where the training dataset is divided into 10 equal

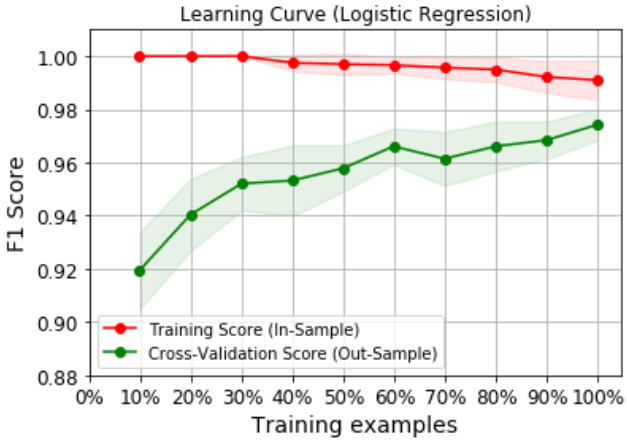
portions of 10% increments. We then added in a title and labels, and color coded the in-sample and out-sample performances with red and green respectively.

In [29]:
```
################################ Define function that plots Learning Curves #################################

def plot_learning_curve(estimator, title, X, y, ylim=None, cv=None,
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 10)): # np.linspace(.1, 1.0, 5) will return evenly
                                                                         # spaced 5 numbers from 0.1 to 1.0
                        # n_jobs is the number of CPUs to use to do the computation.
# Visualization patamters
    plt.figure()                    #display figure
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples") #y label title
    plt.ylabel("F1 Score")             #x label title

    # Estimate train and test score for different training set sizes
    # Class learning_curve determines cross-validated training and test scores for different training set sizes
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes) # learning_curve Determines cross-validated
                                                                        # training and test scores for different
                                                                        # training set sizes.
    # Cross validation statistics for training and testing data (mean and standard deviation)
    # Estimate statistics of train and test scores (mean, std)
    train_scores_mean = np.mean(train_scores, axis=1) # Compute the arithmetic mean along the specified axis.
    train_scores_std = np.std(train_scores, axis=1)   # Compute the standard deviation along the specified axis.
    test_scores_mean = np.mean(test_scores, axis=1)   # Compute the arithmetic mean along the specified axis.
    test_scores_std = np.std(test_scores, axis=1)     # Compute the standard deviation along the specified axis.

    plt.grid() # Configure the grid lines

    # Fill the area around the mean scores with standard deviation info
    # and test data
    plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                     train_scores_mean + train_scores_std, alpha=0.1,
                     color="r") # train data performance indicated with red
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                     test_scores_mean + test_scores_std, alpha=0.1, color="g") # test data performance indicated with green

    # Cross-validation means indicated by dots
    # Train data performance indicated with red
    # Visualization parameters that will allow us to distinguish train set scores from test set scores
    plt.plot(train_sizes, train_scores_mean, 'o-', color="r",
             label="Training Score (In-Sample)")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
             label="Cross-Validation Score (Out-Sample)")

    plt.legend(loc="best") # Show legend of the plot at the best location possible
    return plt             # Function that returns the plot as an output
```

We kept the number of splits for cross validation at 5, test size at 30% and random state at 42 for our test score. We copied over our optimal estimator with the optimized parameters from our parameter tuning back in part a). While we only optimized 2 parameters for logistic regression, coping the rest do not affect the model as they are set to the default anyways.

In [50]:
```
##################################### Plot Learning Curve for Logistic Regression #####################################

# Determines cross-validated training and test scores for different training set sizes
from sklearn.model_selection import learning_curve
# Random permutation cross-validator
from sklearn.model_selection import ShuffleSplit
# Logistic regression classifier class
from sklearn.linear_model import LogisticRegression
# kNN classifier class
from sklearn import neighbors
# Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure,
# plots some lines in a plotting area, decorates the plot with labels, etc

title = "Learning Curve (Logistic Regression)"
# SVC is more expensive so we do a lower number of CV iterations:
# Class ShuffleSplit is a random permutation cross-validator
# Parameter n_splits = Number of re-shuffling & splitting iterations
# Parameter test_size = represents the proportion of the dataset to include in the test split (float between 0.0 and 1.0)
# Parameter random_state = the seed used by the random number generator

cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=42)
estimator = LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l1', random_state=42, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
# Plots the learning curve based on the previously defined function for the logistic regression estimator
plot_learning_curve(estimator, title, X, y, (0.88, 1.01), cv=cv, n_jobs=4, train_sizes=np.linspace(.1, 1.0, 10))
plt.xticks(np.linspace(0, 398, 11), ('0%','10%', '20%', '30%', '40%', '50%','60%','70%','80%','90%','100%'))
plt.show() # Display the figure
```

The resulting graph is below:



To interpret and draw insights from the learning curve, we must first understand that a learning curve shows the generalization performance plotted against the amount of training data used. We should expect performance to change (generally improve) as training sample size increases, to a certain point before flattening out. Typically, models are initially steep as the procedure finds the most apparent regularities in the dataset, but the marginal advantage of more data decreases as the model matures, and performance improvements can no longer be procured. The textbook also mentioned that learning curves have many additional analytical uses. IN real life, additional data often means more cost, so knowing at which point additional investment into more data will not yield meaningful return is crucial. Instead, these companies can then direct their resources into improving the model in other ways such as devising better features.

For our model, we can see that the training f1 score starts off strong but slopes downwards and settles at around 0.99 when we use 100% of our training dataset. On the other hand, our out-sample performance starts off low at around 0.92 but continues to climb until it slows down its marginal benefit at around 0.96 and eventually settles at around 0.975 with 100% training data utilization. This convergence is likely because the in-sample performance at the beginning happens to be good at predicting the performance of the very dataset the model is built on, but the model is in reality not mature, leading to a very poor test f1 score where it makes many false positive and true negative predictions. However, as the model matures with increasing training data, the out-sample f1 score drastically improves as it makes less and less mistakes, while the in-sample f1 score can no longer maintain its perfect score and drops just a tiny bit. It is important to note that this drop between 10% training data and 100% training data is less than 0.01, which is little if any significance. This is similar to a student who had a 4.0 gpa, but had one 3.7, and now has a 3.99 and can never go back to 4.0, but is still a good student regardless.

If I were to be asked to make a call on how much data to use from a business perspective, I would say either 60% or 100% depending on your perspective. If this data is very expensive and there isn't much marginal utility between 0.965 and 0.975 (less than 0.01 difference), then I would say only get 60% training data, as that extra 40% represent a huge cost and not much improvement. On the other hand, in our case where literally lives are on the line (and assuming people value lives more than the cost of the 40% more training data), then I would say definitely get 100% of training data and use all of it, as that 0.01 difference between 60% and 100% performance might mean the lives of thousands and the future of many, many families. Even one less false positive or false negative prediction can result in a lot of less grief and a lot of less deaths respectively.

c) Build a fitting graph for different depths of the decision tree (visualize the performance for both training and test data in the same plot). Please provide screenshots of your code and explain the process you have followed.

[part c is worth 25 points in total:
8 points for correct visualization of fitting graph for in-sample sample performance (training) – show the performance for **15 different values**- provide screenshots of your code and explain the process you have followed
8 points for correct visualization of fitting graph for out-of-sample performance (test)– show the performance for 15 different values- provide screenshots of your code and explain the process you have followed
9 points for discussing what we can learn from this specific fitting graph – what are the insights that can be drawn] – imagine you are talking to another team, how would you describe the insights?

Constructing the Fitting Graph is again very intuitive. We first copy over the optimized model with the tuned parameters. We then remove max_depth, as we want to see how this parameter affects the performance of the rest of our model. Param_range is declared with 15 different values. It is important to note that because we are using cross validation

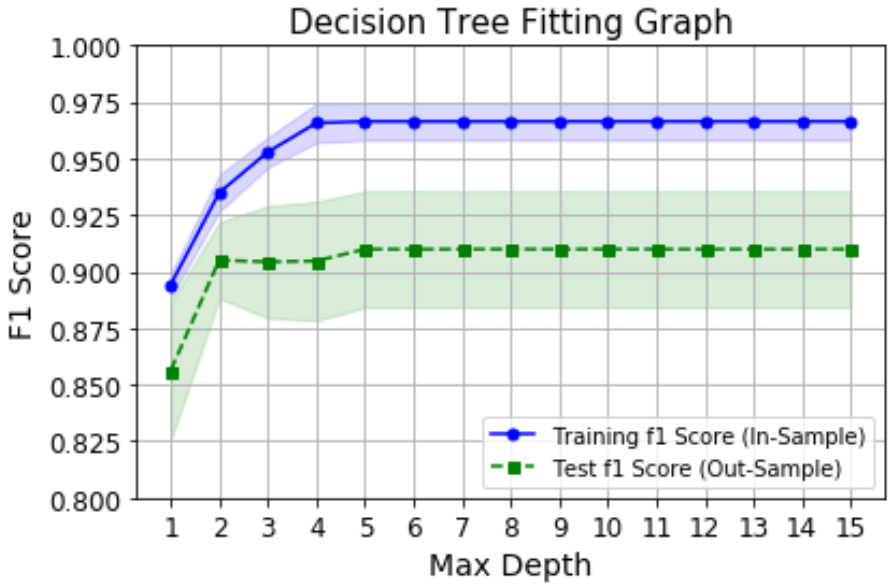
for test performance, we will be using the entire dataset (training and test combined) to illustrate the effect of max_depth on our model. Using Training data as our X means we will be cross validating only the training data, which won't make much sense.

In [56]:
```
############################ Parameters - Varying Complexity ############################
# Specify possible depths for the tree.
param_range = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
# Compute scores for an estimator with different values of a specified parameter.
# This is similar to grid search with one parameter.
# However, this will also compute training scores and is merely a utility for plotting the results.

######################### Estimate Scores - Varying Complexity #########################

# Determine training and test scores for varying parameter values.
train_scores, test_scores = validation_curve(
                #estimator=DecisionTreeClassifier(random_state=42), #Build Logistic Regression Models
                estimator=DecisionTreeClassifier(class_weight=None, criterion='gini',
                max_features=None, max_leaf_nodes=None,
                min_impurity_decrease=0.0, min_impurity_split=None,
                min_samples_leaf=5, min_samples_split=2,
                min_weight_fraction_leaf=0.0, presort=False, random_state=42,
                splitter='best'),
                X=X, #X=X because we are testing the model and not optimizing
                y=y, #
                param_name="max_depth", # parameter C: Inverse of regularization strength; must be a positive float. Smaller val
                param_range=param_range,
                cv=5,     #5-fold cross-validation
                scoring="f1",
                n_jobs=4) # Number of CPU cores used when parallelizing over classes if multi_class='ovr'". This parameter is ig

# Cross validation statistics for training and testing data (mean and standard deviation)
train_mean = np.mean(train_scores, axis=1) # Compute the arithmetic mean along the specified axis.
train_std = np.std(train_scores, axis=1)   # Compute the standard deviation along the specified axis.
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

Cross validation is used to get the test performance, and we plot two lines, one for the training performance and one for the test performance. We adjust the tickmarks to accurately showcase the 15 different depths of our tree.

In [57]:
```
############################## Visualization - Fitting Graph ##############################

# Plot train f1 means of cross-validation for all the parameters C in param_range
plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5, label='Training f1 Score (In-Sample)')

# Fill the area around the line to indicate the size of standard deviations of performance for the training data
plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')

# Plot test f1 means of cross-validation for all the parameters C in param_range
plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='Test f1 Score (Out-Sample)')

# Fill the area around the line to indicate the size of standard deviations of performance for the test data
plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

# Grid and Axes Titles
plt.title('Decision Tree Fitting Graph', loc='center',fontsize=15)
#plt.suptitle('Fitting Graph', loc='center')
plt.grid()
plt.xscale('linear')
plt.legend(loc='lower right')
plt.xlabel('Max Depth')
plt.ylabel('F1 Score')
plt.xticks(np.linspace(1, 15, 15))
plt.yticks(np.linspace(0.8, 1, 9))
#plt.ylim([0.8, 1.01]) # y limits in the plot
plt.tight_layout()
# plt.savefig('Fitting_graph_LR.png', dpi=300)
```

The resulting fitting graph is below:



Fitting Graphs are useful in improving model performance (f1 score in this case) while only adjusting a single parameter. This is especially useful in addressing over and underfitting, as error rate typically will increase on out-sample as a model becomes overfit, while the error rate on in-sample will approach 0 as the model memorizes all the features of the training set. While fitting graphs are related to learning curves, they are very different, as learning curves show performance of a model with a set of parameters at different test sample sizes, while fitting graphs show the performance of a model with a fixed training set size as one parameter changes, in our case the max depth of our decision tree. Similar to the learning curve example above, we plotted the average training and cross-validation accuracies and the corresponding standard deviations. Although the differences in the accuracy for varying values of max_depth is very subtle, we can definitely see that both training and test f1 scores make a big jump from 1 to 2. While Test f1 score seems to slow down ad level off from 2 to 4, training f1 score continues to improve until it hits a plateau at around 0.970. As a judgement call, I think the model slightly underfits the data when we can only have 1 as the max_depth, which makes a lot of sense as we limit the number of layers to our tree. Interestingly, increasing number of max_depth does not result in any indications of overfitting. I think this is the case because max_depth **limits** the number of layers our model **can have** but does not actually set the number of layers. As our dataset is quite simple and our model isn't too complex, 2-5 layers is more than enough to develop a mature model. Setting another parameter for our fitting graph should yield more interesting results on another parameter that is not a limit, or on a model or dataset that is a lot more complex and has a lot of more parameters and variables respectively.

However, two layers seems to be the beginning of where we should consider setting the max_depth at, with 5 being the absolute max we should consider setting our max_depth at as any additionally layers only make our tree more complex without increasing any test f1 performance. As our goal is to discuss our model with stakeholders who may not be the most adept at understanding complex models, we should choose 2 for max depth as this keeps our model very clean and simple to understand without requiring a lot of technical knowledge. On the other hand, as I have mentioned in my answer for b, if our goal is to maximize the number of lives saved by minimizing the number of false prediction, we should choose 5 for our max depth as that small increase in f1 score could mean a lot of less false positive and negative predictions in reality and many lives and families saved. Regardless, both 2 and 5 result in respective f1 scores of around 0.91-0.92, and the increase is very marginal.

d) Create an ROC curve for k-NN, decision tree, and logistic regression. Discuss the results. Which classifier would you prefer to choose? Please provide screenshots of your code and explain the process you have followed.

[part d is worth 25 points in total:
5 points for correct visualization of ROC graph for kNN – use optimal kNN from part a
5 points for correct visualization of ROC graph for Decision Tree – use optimal Decision Tree from part a
5 points for correct visualization of ROC graph for Logistic Regression – use optimal Logistic Regression from part a
2 points for showing all the ROC graphs in one single plot
3 points for showing AUC estimators in the ROC graph

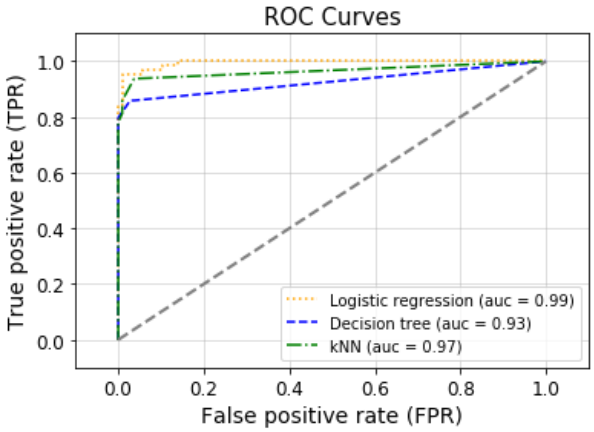
5 points for discussing and correctly identifying which classifier you would use]- optimal

The last part is the ROC curve for the 3 models. First, we copied all the optimized models with the tuned parameters from 3 a), and set them as clf1, clf2 and clf3. As I have stated above, the parameters that we did not tune were simply listed as their default, so copying them over would not affect our model in any way.

In [18]:
```
################################### Classifiers ####################################

#Normalize Data
#sc = StandardScaler()
#sc.fit(X_train)
#X_train_std = sc.transform(X_train)
#X_test_std = sc.transform(X_test)

#Normalize Data
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

# Logistic Regression Classifier
clf1 = LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l1', random_state=42, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)

# Decision Tree Classifier
clf2 = DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')

# kNN Classifier
clf3 = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=1,
           weights='uniform')

# Label the classifiers
clf_labels = ['Logistic regression', 'Decision tree', 'kNN']
all_clf = [clf1, clf2, clf3]
```

```
################################### Cross - Validation ###################################
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

print('5-fold cross validation:\n')
# Note: We are assuming here that the data is standardized. For the homework, you need to make sure the data is standardized.
for clf, label in zip([clf1, clf2, clf3], clf_labels): #For all classifiers
    scores = cross_val_score(estimator=clf,  #Estimate AUC based on cross validation
                             X=X_std, #We are using standardized dataset
                             y=y,
                             cv=5,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]" #Print peformance statistics based on cross-validation
          % (scores.mean(), scores.std(), label))

################################### Visualization ####################################

colors = [ 'orange', 'blue', 'green']      # Colors for visualization
linestyles = [':', '--', '-.', '-']        # Line styles for visualization
for clf, label, clr, ls in zip(all_clf,
               clf_labels, colors, linestyles):

    # Assuming the label of the positive class is 1 and data is normalized
    y_pred = clf.fit(X_train, #use std?
                     y_train).predict_proba(X_test)[:, 1] # Make predictions based on the classifiers
    fpr, tpr, thresholds = roc_curve(y_true=y_test, # Build ROC curve
                                     y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)                # Compute Area Under the Curve (AUC)
    plt.plot(fpr, tpr,                         # Plot ROC Curve and create label with AUC values
             color=clr,
             linestyle=ls,
             label='%s (auc = %0.2f)' % (label, roc_auc))

plt.legend(loc='lower right')    # Where to place the legend
plt.plot([0, 1], [0, 1], # Visualize random classifier
         linestyle='--',
         color='gray',
         linewidth=2)

plt.xlim([-0.1, 1.1])   #Limits for x axis
plt.ylim([-0.1, 1.1])   #Limits for y axis
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')


#plt.savefig('ROC_all_classifiers', dpi=300)
plt.show()
```

```
5-fold cross validation:

ROC AUC: 0.98 (+/- 0.01) [Logistic regression]
ROC AUC: 0.95 (+/- 0.02) [Decision tree]
ROC AUC: 0.99 (+/- 0.01) [kNN]
```



To interpret and compare/contrast the three graphs, let's first take a look at what the x (FPrate) and y (TPrate) axes mean.

$$FP_{rate} = \frac{FP}{N} = \frac{FP}{FP+TN}$$ ← Fraction of negative examples incorrectly classified

$$TP_{rate} = \frac{TP}{P} = \frac{TP}{TP+FN}$$ ← Fraction of positive examples correctly classified (recall)

With any model, we want to maximize True positive and true negatives while minimizing false positive and false negatives. With this said, we want to maximize tp rate, which indicate the smallest number of false negatives possible and minimize tp rate, which indicates the smallest number of false positives. Thus, the perfect model would reach the point (0.0, 1.0) on the graph above. Of all the three models, it is clear that the logistic regression model is the most optimized model and performs the best, as the point on the logistic regression curve that is closest to the (0,1) point is closer than that of the other two models. In our case, where a false negative prediction is much more costly than a false positive prediction (cancer gone undetected vs traumatized family), increase in y is valued a lot more than increase in x. Luckily, we don't have to compare and contrast different models in this regard, as the ROC curve dominates. **TLDR: the Logistic Regression Curve is the best by far.**

Because kNN requires standardized data but the other two do not, I have tried using both standardized data and raw data for cross-validation and the graph, but both resulted in the same graph. In addition, as we are trying to compare across the three models, it shouldn't be too big of an issue as long as all three uses the same X dataset. With this being said, professor Todri explicitly stated in the cross-validation module below that *"We are assuming here that the data is standardized. For the homework, you need to make sure the data Is standardized."* To be sure, I will be discussing the results from my unstandardized call. But I also reran parameter optimization calls for all three models and used the models with the parameters I got from that for my ROC curve, again using standardized X.

Model and nested cv f1-score stayed the same for the decision tree with standardized X:

```
############################## Decision Tree Parameter Tuning ###############################

# See all the parameters you can optimize here http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassif
# Choosing optimal depth of the tree
# Choosing optimal depth of the tree AND optimal splitting criterion
# Choosing depth of the tree AND splitting criterion AND min_samples_leaf AND min_samples_split
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

gs_dt = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42),
                     param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7,8, None], 'criterion':['gini','entropy'],
                                  'min_samples_leaf':[1,2,3,4,5],
                                  'min_samples_split':[2,3,4,5]}],
                     scoring='f1', # Specifying multiple metrics for evaluation
                     cv=inner_cv,
                     n_jobs=-1)

gs_dt = gs_dt.fit(X_std,y)
print("\nDecision Tree Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_dt.best_score_)
print("Optimal Parameter: ", gs_dt.best_params_) # Parameter setting that gave the best results on the hold out data.
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest
nested_score_gs_dt = cross_val_score(gs_dt, X=X_std, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
Decision Tree Parameter Tuning
Non-nested CV F1-Score:  0.934657984394364
Optimal Parameter:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 5, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')
Nested CV F1-Score:  0.9265708104408736  +/-  0.014452515617888772
```

Model and nested cv f1-score both changed for logistic regression with standardized X:

```
############################ Logistic Regression Parameter Tuning ############################
# Choosing C parameter (i.e., regularization parameter) for Logistic Regression
# Choosing C parameter for Logistic Regression AND type of penalty (ie., l1 vs l2)
# See other parameters here http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

gs_lr = GridSearchCV(estimator=LogisticRegression(random_state=42),
                  param_grid=[{'C': [ 0.00001, 0.0001, 0.001, 0.01, 0.1 ,1 ,10 ,100, 1000, 10000, 100000, 1000000, 10000000],
                               'penalty':['l1','l2']}],
                  scoring='f1',
                  cv=inner_cv)

gs_lr = gs_lr.fit(X_std,y)
print("\n Logistic Regression Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_lr.best_score_)
print("Optimal Parameter: ", gs_lr.best_params_)
print("Optimal Estimator: ", gs_lr.best_estimator_)
nested_score_gs_lr = cross_val_score(gs_lr, X=X_std, y=y, cv=outer_cv)
print("Nested CV F1-Score:",nested_score_gs_lr.mean(), " +/- ", nested_score_gs_lr.std())
```

```
 Logistic Regression Parameter Tuning
Non-nested CV F1-Score:  0.9741332753342583
Optimal Parameter:  {'C': 0.1, 'penalty': 'l2'}
Optimal Estimator:  LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=42, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
Nested CV F1-Score: 0.9741006828434184  +/-  0.012735348788663165
```

Model and nested cv f1-score both changed for logistic regression with standardized X:

kNN Parameter Tuning

In [68]:

```
#We Tune our parameters using the entire standardized X set
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

# Choosing k for kNN
# Choosing k for kNN AND type of distance
gs_knn = GridSearchCV(estimator=neighbors.KNeighborsClassifier(
                         metric='minkowski'),
                  param_grid=[{'n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
                               'weights':['uniform','distance'],'p':[1,2,3,4,5,6,7,8,9,10]}],
                  scoring='f1',
                  cv=inner_cv,
                  n_jobs=4)

#print(len(y))
gs_knn = gs_knn.fit(X_std,y)  #X_std
print("\n kNN Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_knn.best_score_)
print("Optimal Parameter: ", gs_knn.best_params_)
print("Optimal Estimator: ", gs_knn.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highes
nested_score_gs_knn = cross_val_score(gs_knn, X=X_std, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
```

```
 kNN Parameter Tuning
Non-nested CV F1-Score:  0.9574747147707873
Optimal Parameter:  {'n_neighbors': 9, 'p': 2, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=9, p=2,
           weights='uniform')
Nested CV F1-Score:  0.9464914636342086  +/-  0.03280761754947194
```

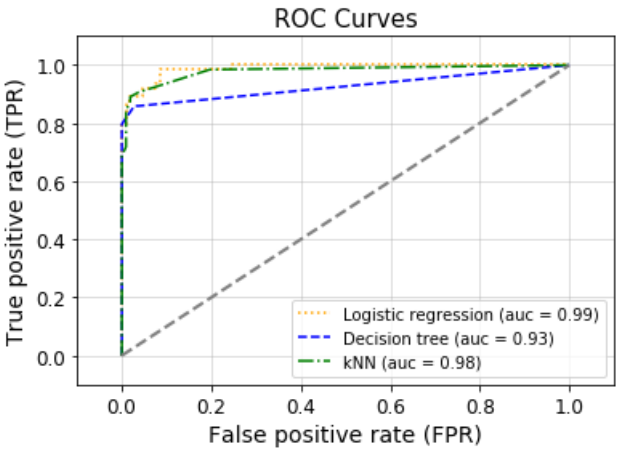
Recreating ROC Curves with standardized model and X:

In [69]:

```
##################################### Classifiers ######################################

#Normalize Data
#sc = StandardScaler()
#sc.fit(X_train)
#X_train_std = sc.transform(X_train)
#X_test_std = sc.transform(X_test)

#Normalize Data
sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)

# Logistic Regression Classifier
clf1 = LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=42, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)

# Decision Tree Classifier
clf2 = DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=42,
            splitter='best')

# kNN Classifier
clf3 = KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=9, p=2,
           weights='uniform')

# Label the classifiers
clf_labels = ['Logistic regression', 'Decision tree', 'kNN']
all_clf = [clf1, clf2, clf3]

################################### Cross - Validation ##################################
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

print('5-fold cross validation:\n')
# Note: We are assuming here that the data is standardized. For the homework, you need to make sure the data is standardized.
for clf, label in zip([clf1, clf2, clf3], clf_labels): #For all classifiers
    scores = cross_val_score(estimator=clf,  #Estimate AUC based on cross validation
                             X=X_std, #We are using standardized dataset
                             y=y,
                             cv=5,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]" #Print peformance statistics based on cross-validation
          % (scores.mean(), scores.std(), label))
```

```
##################################### Visualization #####################################

colors = [ 'orange', 'blue', 'green']      # Colors for visualization
linestyles = [':', '--', '-.', '-']        # Line styles for visualization
for clf, label, clr, ls in zip(all_clf,
               clf_labels, colors, linestyles):

    # Assuming the label of the positive class is 1 and data is normalized
    y_pred = clf.fit(X_train, #use std?
                     y_train).predict_proba(X_test)[:, 1] # Make predictions based on the classifiers
    fpr, tpr, thresholds = roc_curve(y_true=y_test, # Build ROC curve
                                     y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)                # Compute Area Under the Curve (AUC)
    plt.plot(fpr, tpr,                         # Plot ROC Curve and create label with AUC values
             color=clr,
             linestyle=ls,
             label='%s (auc = %0.2f)' % (label, roc_auc))

plt.legend(loc='lower right')    # Where to place the legend
plt.plot([0, 1], [0, 1], # Visualize random classifier
         linestyle='--',
         color='gray',
         linewidth=2)

plt.xlim([-0.1, 1.1])   #Limits for x axis
plt.ylim([-0.1, 1.1])   #Limits for y axis
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')


#plt.savefig('ROC_all_classifiers', dpi=300)
plt.show()
```

```
5-fold cross validation:

ROC AUC: 1.00 (+/- 0.00) [Logistic regression]
ROC AUC: 0.95 (+/- 0.02) [Decision tree]
ROC AUC: 0.99 (+/- 0.01) [kNN]
```



The Logistic Regression model now performs very close to the kNN model. I have discussed how to evaluate the ROC curve in the unstandardized discussions above. If I had to choose one, I would still choose the logistic regression model. Logistic regression has the point closest to (0.0,1.0), the best scenario that indicates the smallest number of both false negatives and false positives.