EMORY

GOIZUETA
BUSINESS
SCHOOL

# MSBA Team 7 Homework 3b

**Janvier Nshimyumukiza, Ashish Sangai, Sherraina Song, Wenqi Zhai**

**Introduction to Business Analytics**

**September 23, 2022**

# Part a): Parameter Tuning in Python

Before we started to tune the three models, we did some data preprocessing. First, we loaded all the libraries and we read the csv file using read_csv() in python, and encoded the labels just as we did for the last assignment.

```
##################################### Load Libraries and Modules #####################################
#LATEST MODELING CODE AT THE END OF THE SCRIPT! @09/22 BY SUMMER

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split # splits arrays or matrices into random train and test subsets
from sklearn.metrics import accuracy_score, f1_score, classification_report # the sklearn.metrics module includes performance metrics
from sklearn.linear_model import LogisticRegression       # the sklearn.linear_model module implements generalized linear models. LogReg is part of this module
from sklearn.tree import DecisionTreeClassifier
```

```
[42] df = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data', header=None)
```

```
[43] from sklearn import preprocessing
     le = preprocessing.LabelEncoder()
     # Encode the labels
     le.fit(['B','M'])
     # Show the label classes
     print(le.classes_)
     # Transform the label of the table
     df[1] = le.transform(df[1])
```

```
['B' 'M']
```

And we defined the target variable, whether the cancer is benign or malignant as the y and set the X to be the 30 columns besides target variable and the id.

```
y = df[1]                              # Column one 0: Benign, and 1: Malignant
X = df.loc[:, 2:  ]                 # Select all columns except column 0 and 1
```

For cross validation, we set the folds number to 5 for both the inner and outer split. When that is done, we moved on to part A and tuned the three models by optimizing the parameters.

```
inner_cv = KFold(n_splits=5, shuffle=True) # inner cross-validation folds
outer_cv = KFold(n_splits=5, shuffle=True) # outer cross-validation folds
```

### 1) Optimizing the parameters for the Decision Tree

We imported the needed libraries and to ensure reproducibility of our model, we set the random.seed to 42 for the decision tree.

```
############################## Import Libraries & Modules ###############################
from sklearn.tree import DecisionTreeClassifier # A decision tree classifier
# GridSearchCV performs an exhaustive search over specified parameter values for an estimator
# The parameters of the estimator used to apply these methods are optimized by cross-validated
# grid-search over a parameter grid.
# Documentation: http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
from sklearn.model_selection import GridSearchCV, KFold, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn import neighbors, datasets
# Standardize features by removing the mean and scaling to unit variance
from sklearn.preprocessing import StandardScaler
# Pipeline of transforms with a final estimator
from sklearn.pipeline import Pipeline
import numpy as np
```

```
np.random.seed(42) # ensure reproducability
```

The four parameters we optimized for decision tree are:

1) Max_depth: We want to see the performance for the maximum depth of the tree. By changing this parameter's value range from 1-7 and none. Optimizing this parameter is very critical because when k is too large, it's likely that the model will overfit.
2) Criterion: whether using gini or entropy as the splitting criterion
3) Min_samples_leaf: we want to see from 1-5, which yields the best performance as the minimum number of samples required to be at a leaf node.
4) Min_samples_split: we want to see which is the minimum number of samples required to split an internal node

Then we passed the predefined values for hyperparameters to the GridSearchCV function from the sklearn.model_selection module. We used the decision tree as the estimator and random state of the DecisionTreeClassifier is set to 42 again to ensure the same result can be reproduced. We use f1-score as the metric for evaluation since this is what we believed the most appropriate metric based on the last project, and inner_cv specified above as the cross-validation folds and fit the model using fit().

Corresponding code

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

# Choosing depth of the tree AND splitting criterion AND min_samples_leaf AND min_samples_split
gs_dt2 = 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',
                  cv=inner_cv,
                  n_jobs=4)

gs_dt2 = gs_dt2.fit(X,y)
print("\n Parameter Tuning(max_depth, criterion, min_smaples_leaf, min_samples_split) for Decision Tree")
print("Non-nested CV F1-Score: ", gs_dt2.best_score_)
print("Optimal Parameter: ", gs_dt2.best_params_)
print("Optimal Estimator: ", gs_dt2.best_estimator_)
nested_score_gs_dt2 = cross_val_score(gs_dt2, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_dt2.mean(), " +/- ", nested_score_gs_dt2.std())
```

Below is the result for the decision tree model after we used the training data to perform the grid search. Using best_score_ attribute we obtained the score of the best-performing model that is 93.47%. The optimal parameters setting gave us information about the best results on the hold out data and decision tree: using splitting criterion gini and set max_depth as 5, min_sample_leaf as 5 and min_samples_split as 2. The estimator gave the highest f1 score when max_depth is 5 and min_sampels__leaf is 5 and the random_State is 42. The optimal f1 score from the best decision tree model in the outer cross-validation is ~91.48%.

Corresponding result

```
 Parameter Tuning(max_depth, criterion, min_smaples_leaf, min_samples_split) for Decision Tree
Non-nested CV F1-Score:  0.9346928746928747
Optimal Parameter:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 5, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeClassifier(max_depth=5, min_samples_leaf=5, random_state=42)
Nested CV F1-Score:  0.9147744492316287  +/-  0.024981294787751095
```

**2) Optimizing the parameters for the Logistic Regression**

The two parameters we are optimizing for the logistic regression model is C and type of penalty:

1) C: C is the inverse of the regularization factor. Larger values of parameter C specify weaker regularization and we chose it because the strength for regularization has a huge impact on the prediction of the logistic regression model. We imputed those 13 values ranging from 0.00001, 0.0001, 0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000 into the dictionary for C.
2) Type of penalty: We used it because we wanted to check whether using L1-norm or L2 norm is most appropriate for breast cancer data.

Corresponding Code

```
########################## Logistic Regression Parameter Tuning ############################
#To ignore the convergence warnings
from  warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
simplefilter("ignore", category=ConvergenceWarning)

np.random.seed(42) # ensure reproducability

# 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_lr2 = GridSearchCV(estimator=LogisticRegression(random_state=42, solver='liblinear'),
                  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_lr2 = gs_lr2.fit(X,y)
print("\n Parameter Tuning(C, penalty) for Logistic Regression")
print("Non-nested CV F1-Score: ", gs_lr2.best_score_)
print("Optimal Parameter: ", gs_lr2.best_params_)
print("Optimal Estimator: ", gs_lr2.best_estimator_)
nested_score_gs_lr2 = cross_val_score(gs_lr2, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score:",nested_score_gs_lr2.mean(), " +/- ", nested_score_gs_lr2.std())
```

Like we did for the decision tree, we passed the parameters to the GridSearchCV to train and tune the model then fitted the model using X and y. The result shows that the best parameter generated

is C equals to 100, which is the inverse of the regularization parameter and uses L1-norm as the penalty. The best estimator is when c=100, penalty is L1, random_state set to 42 and solver set as liblinear. The corresponding F-1 score for the Logistic Regression classifier is ~93.43%. This is better than the F-1 score from the decision tree.

Corresponding Result

```
 Parameter Tuning(C, penalty) for Logistic Regression
Non-nested CV F1-Score:  0.9576785075389728
Optimal Parameter:  {'C': 100, 'penalty': 'l1'}
Optimal Estimator:  LogisticRegression(C=100, penalty='l1', random_state=42, solver='liblinear')
Nested CV F1-Score: 0.9342980086462273  +/-  0.027574016117070804
```

### 3) Optimizing the parameters for the kNN

We built up the pipeline to first standardize the data using StandardScaler() and use p=2 and using minkowski to calculate the Euclidean distance for the kNN model. Again by initiating the GridSearchCV, we want to train and tune the logistic regression model. We set the param_grid to the two parameters that we want to optimized and passed them as list of dictionaries:

1) knn__n_neighbors: We chose it because k stands for the number of nearest neighbors. The input is all odd numbers for k ranges from 1-21.

2) knn__weights: this is the type of distance for the model. The input is uniform or distance. We tuned this because whether all points in each neighborhood are weighted equally or by the inverse of their distance can impact the performance.

Corresponding Code

```
################################# kNN Parameter Tuning ################################

#Normalize Data
pipe = Pipeline([
       ('sc', StandardScaler()),
       ('knn', KNeighborsClassifier(p=2,
                                    metric='minkowski'))
     ])


#Parameters to optimize:  k for number of nearest neighbors AND type of distance

params = {
       'knn__n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
       'knn__weights': ['uniform', 'distance']
    }

gs_knn2 = GridSearchCV(estimator=pipe,
                  param_grid=params,
                  scoring='f1',
                  cv=inner_cv,
                  n_jobs=4)

gs_knn2 = gs_knn2.fit(X,y)
print("\n Parameter Tuning(knn__n_neighbors,knn__weights) for kNN")
print("Non-nested CV F1-Score: ", gs_knn2.best_score_)
print("Optimal Parameter: ", gs_knn2.best_params_)
print("Optimal Estimator: ", gs_knn2.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest score
nested_score_gs_knn2 = cross_val_score(gs_knn2, X=X, y=y, cv=outer_cv)
print("Nested CV F1-Score: ",nested_score_gs_knn2.mean(), " +/- ", nested_score_gs_knn2.std())
```

Result

We obtained the best value for the k for the number of nearest neighbors is 9 and the and the best knn__weights (type of distance) is using a uniform that yielded the best F-1 score for the kNN algorithm. We use the independent test dataset to estimate the performance of the best selected model, which we got via the best_estimator_ attribute that is k=9 for the neighbors' number using the standardized data and the corresponding F-1 score for the kNN classifier from the nested Cross Validation is ~94.93% with a standard deviation of ~0.028.

```
 Parameter Tuning(knn__n_neighbors,knn__weights) for kNN
Non-nested CV F1-Score:  0.9523682840047905
Optimal Parameter:  {'knn__n_neighbors': 9, 'knn__weights': 'uniform'}
Optimal Estimator:  Pipeline(steps=[('sc', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=9))])
Nested CV F1-Score:  0.9492982074989029  +/-  0.017757982733385847
```

### 4) Comparison between the performance of the three models

We compared the generalization performance of the three classifiers using the F-1 score shown as Nested CV Accuracy in our code snippet. The optimal parameters for Decision tree is using splitting criterion gini and set max_depth as 5, min_sample_leaf as 5 and min_samples_split as 2 and the F-1 score generated from nested cross validation is ~91.48%. The optimal parameters for logistic regression is when C equals 100 and L1-norm as the penalty, and it yields a F-1 score of ~93.43%. for kNN is ~94.93%.

Because the tuned kNN classifier yielded the highest F1-score, we chose kNN when k=9 and knn__weights = 'uniform' as the model we preferred to use because it generated higher value for both recall and precision and we wanted to maximize them and took both false positives and false negatives into account.

# b) Logistic Regression Learning Curve for the Tumor Classification Data

In this subsection, we built and visualized a learning curve for the logistic regression technique. The visualization of the performance is presented for both training and test data in the same plot. And below are the screenshots of our code.

```
[41] ############################ Function for Learning Curves ############################

     def plot_learning_curve(estimator,       # data science algorithm
                             title,           # title of the plot
                             X, y,            # data (features and target variable)
                             ylim=None,       # minimum and maximum y values plotted
                             cv=None,         # cross validation splits
                             n_jobs=1,        # parallell estimation using multiple processors
                             train_sizes=np.linspace(.1, 1.0, 10)): #linspace returns evenly spaced numbers over a specified interval (start, stop, num)
```

```
    plt.figure()                  # display figure
    plt.title(title)              # specify title based on parameter provided as input
    if ylim is not None:          # if ylim was specified as an input, make sure the plots use these limits
        plt.ylim(*ylim)
    plt.xlabel("Training examples") # y label title
    plt.ylabel("Score")           # x label title

    # Class learning_curve determines cross-validated training and test scores for different training set sizes
    train_sizes, train_scores, test_scores = learning_curve(estimator, # data science algorithm
                                                            X, y,      # data (features and target variable)
                                                            cv=cv,     # cross-validation folds
                                                            n_jobs=n_jobs, # number of jobs to run in parallel using multiple processors
                                                            train_sizes=train_sizes) # relative or absolute numbers of training examples
                                                                                     # that will be used to generate the learning curve

    # Cross validation statistics for training and testing data (mean and standard deviation)
    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 in the plot

    # Fill the area around the line to indicate the size of standard deviations for the training data
    # and the test data
    plt.fill_between(train_sizes,                          # the x coordinates of the nodes defining the curves
                     train_scores_mean - train_scores_std, # the y coordinates of the nodes defining the first curve
                     train_scores_mean + train_scores_std, # the y coordinates of the nodes defining the second curve
                     alpha=0.1,                            # level of transparency in the color fill
                     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
    plt.plot(train_sizes,                                    # the horizontal coordinates of the data points
             train_scores_mean,                              # the vertical coordinates of the data points
             'o-',                                           # o- will produce a small circle and a solid line to connect the markers
             color="r",                                      # line of red color
             label="Training score")                         # specify label title for this plot

    # Test data performance indicated with green
    plt.plot(train_sizes,
             test_scores_mean,
             'o-',
             color="g",                                      # line of green color
             label="Cross-validation score")

    plt.legend(loc="best")                    # show legend of the plot at the best location possible
    return plt
```

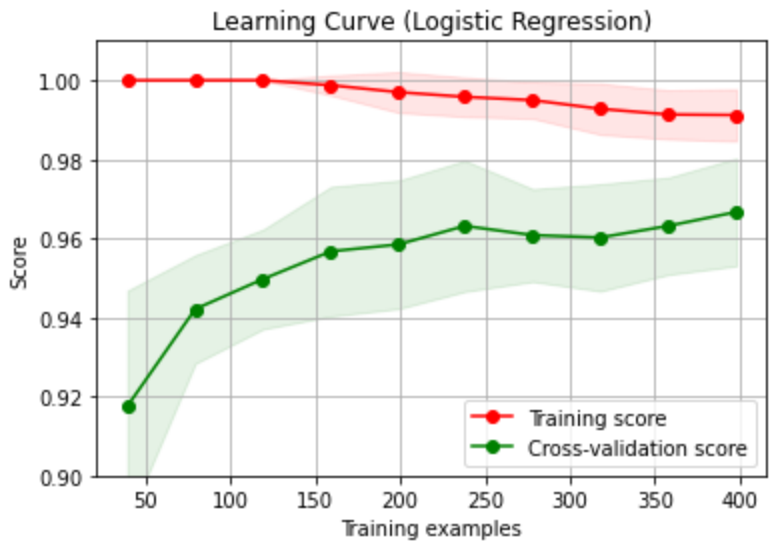
In reference to the codes provided within the hyper-parameters tuning and model optimization lectures material in class, we have defined the function above (plot_learning_curve). This function receives the model(named estimator), the data, the title of the plot, and other arguments that customize the graph.

This function divides the data into train and test data and trains the model with cross-validation with 10-Folds in the data. It used sklearn's learning curve which trained 10 models for a different number of observations. Starting with fewer observations and gradually increasing the number of observations in the training data. This allowed us to assess if getting more data would help us build a better model.

We called this self-defined function plot_learning_curve() to generate the plot by first passing estimator, that is LogisticRegression() because we wanted to build multiple LR models and compared their performance using different sizes of training data; and we passed predefined title, X, y, the minimum and maximum y values plotted (0.9, 1.01) , the same fold we used from part a and the parallel estimation using multiple processors is 4.

```
title = "Learning Curve (Logistic Regression)"

# Class ShuffleSplit is a random permutation cross-validator
cv = ShuffleSplit(n_splits=10,          # number of re-shuffling & splitting iterations
                  test_size=0.3,        # represents the proportion of the dataset to include in the test split (float between 0.0 and 1.0)
                  random_state=42)      # the seed used by the random number generator
estimator = LogisticRegression(C=100, penalty='l1', random_state=42, solver='liblinear')    # build multiple LR models as we increase the size of the traning data
                                                                                            # Note that we used the optimal LR parameters we got from the GridSearch

# Plots the learning curve based on the previously defined function for the logistic regression estimator
plot_learning_curve(estimator,      # data science algorithm
                    title,          # title of the plot
                    X, y,           # data (features and target variable)
                    (0.9, 1.01),    # minimum and maximum y values plotted
                    cv=cv,          # cross-validation folds (produced above)
                    n_jobs=4)       # parallell estimation using multiple processors

plt.show()                          # display the figure
```

Below is the visualization of the learning curve for in-sample performance(red) and out-sample sample performance(green) for logistic regression. The graph below shows the performance for 10 different sizes of data.




**Discussion of the learning curves**

From the above learning curve, we can see that the in-sample learning curve is downward-trend. This means that as we get more data, the logistic regression is likely to perform slightly poorly in the training data. This doesn't mean that it's a bad thing for the model because the model is likely to overfit the data if the data size is not big enough.

On the other hand, by looking at the out-of-sample learning curve, f1-score has a clear and consistent positive correlation with the size of training data. This gives insight that getting more data would be beneficial to get the model which generalizes well to unforeseen observations. We conclude that these learning curves tell us that there is room for model improvement if we get more data/observations.

# c) Fitting Graph for Different Depths of the Decision Tree

Within this subsection, we built the fitting graph for different depths of the decision tree and we visualized the performance for both training and test data in the same plot. Below are the screenshots of our code and we elaborate on the process we followed to get the presented results.

In this section, we imported the necessary libraries and classes of sklearn. Most important validation_curve which determines training and test scores for the varying parameter (which controls the model complexity) max_depth values. After that, we specified possible parameter values for the max_depth of the tree, and we defined the estimator as a Decision tree. For the other parameters, we used the ones with the most effective/optimal parameters from GridSearch in

question (a): min_samples_leaf =5 and gini as the splitting criterion and min_samples_split =2. We have also specified our performance metrics as f1-score because this dataset is not balanced.

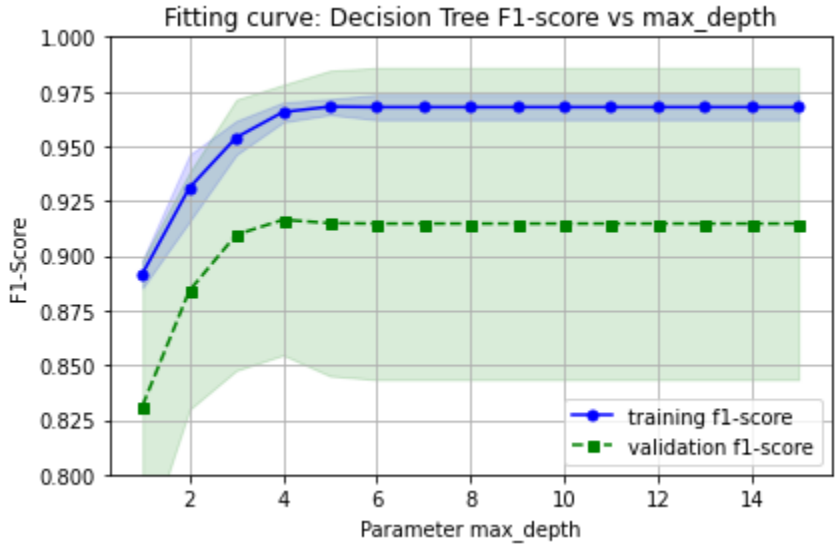
Below are screenshots of our code:

```
# Fitting curve (aka validation curve)
# Determine training and test scores for varying parameter values.
from sklearn.model_selection import validation_curve


np.random.seed(42)                      # the seed used by the random number generator for np

############################ Parameters - Varying Complexity ############################

# Specify possible parameter values for max_depth of the tree.
param_range = range(1,16)

# 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 #########################

train_scores, test_scores = validation_curve(
                estimator=DecisionTreeClassifier(min_samples_leaf=5, criterion='gini', min_samples_split=2, random_state=42), # build Decision Tree models
                X=X_train,                                          # data (features)
                y=y_train,                                          # target variable
                param_name="max_depth",                     # parameter max_depth: enforce the maximum deth of the decision tree model;
                param_range=param_range,                    # the values of the parameter that will be evaluated
                cv=10,                                      # 10-fold cross-validation
                scoring="f1",                               # evaluation metric
                n_jobs=-1)


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

```
############################## Visualization - Fitting Graph ##############################

# Plot train accuracy 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')

# Fill the area around the line to indicate the size of standard deviations of performance for the training data
plt.fill_between(param_range,             # the x coordinates of the nodes defining the curves
                 train_mean + train_std,  # the y coordinates of the nodes defining the first curve
                 train_mean - train_std,  # the y coordinates of the nodes defining the second curve
                 alpha=0.15,              # level of transparency in the color fill
                 color='blue')            # aesthetic parameter - color

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

# 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.grid()
plt.title("Fitting curve: Decision Tree F1-score vs max_depth")
plt.legend(loc='lower right')
```

```
plt.xlabel('Parameter max_depth')
plt.ylabel('F1-Score')
plt.ylim([0.8, 1.0]) # y limits in the plot
plt.tight_layout()
# plt.savefig('Fitting_graph_LR.png', dpi=300)
plt.show()                            # display the figure
```

The following visualization of the fitting graph for in-sample and out-of-sample performance – shows the performance of the decision tree model for 15 different values of the maximum depths enforced to the model.



In the visualization above, in-sample performance is marked by blue and out-of-sample performance is marked by green.

**What Fitting Graph Says**

We applied 15 different maximum depth values to the decision tree model. We visualized the performance of the different values and the results presented above say that max_depth = 4 is the most convenient maximum depth of the tree.

Below 4, the model doesn't give the best performance for both in-sample and out-of-sample data and was likely to be underfitted, and the max depth with values above 4 would increase the complexity of the model while adding no value to the performance of the model for either in-sample or out-of-sample observations. This suggests that increasing the max depth value further has caused a slight overfitting problem. In this case, the sweet spot appears to be around max_depth = 4.

# d) Create a ROC curve for k-NN, Decision Tree, and Logistic Regression

ROC gives us important information with respect to the false positive and true positive rates for the three models and AUC (Area Under the Curve) characterize the performance of the three classification models. To create the ROC curve, firstly we imported the libraries and modules in order to do the ROC and AUC calculations and graphing.

```
############################## Import Libraries & Modules ##################################

import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
```

We loaded all three of the classifiers and made sure all the parameters aligned with part A. We made sure to set all random_state to 42 to ensure the same result can be reproduced and included the optimal parameters for the three models.

```
##################################### Classifiers ######################################
# Logistic Regression Classifier
clf1 = LogisticRegression(penalty='l1',
                          C=100,
                          random_state=42,
                          solver='liblinear')

# Decision Tree Classifier
clf2 = DecisionTreeClassifier(max_depth=5,
                              criterion='gini',
                              min_samples_leaf=5,
                              min_samples_split=2,
                              random_state=42)

# kNN Classifier

clf3 = Pipeline([
        ('sc', StandardScaler()),
        ('knn', KNeighborsClassifier(n_neighbors=9,
                                     p=2,
                                     metric='minkowski'))
      ])

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

################################### Cross - Validation ###################################


print('10-fold cross validation:\n')
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,
                             y=y,
                             cv=10,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]" # print peformance statistics based on cross-validation
          % (scores.mean(), scores.std(), label))
```

The parameters for Logistic Regression are set to penalty of l1, C of 100, and random_state=42, and a solver of “liblinear”. The Decision tree is set to a maximum depth of 5, gini criterion, a minimum samples leaf of 5, a minimum samples split of 2, and a random state of 42 of course. For knn, we standardized the data and then set knn__n_neighbors = 9 and knn__weights = uniform.

Then we passed the estimators as input along with the X, y, and scoring method this time set to roc_auc to the 10-fold cross validation to estimate the AUC.

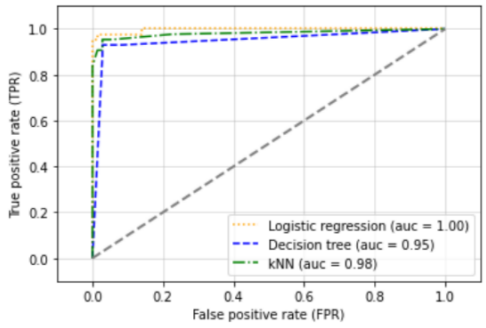
```
10-fold cross validation:

ROC AUC: 0.99 (+/- 0.01) [Logistic regression]
ROC AUC: 0.95 (+/- 0.03) [Decision tree]
ROC AUC: 0.99 (+/- 0.01) [kNN]
```

The Logistic regression and kNN is generated similar AUC performance from the cross validation, and are obviously better than the performance of decision tree.

```
################################### 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, 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='--',         # aesthetic parameters
         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()
```

We created a visualization of the logistic regression, decision tree, kNN models on the same graph using data  from the train_test_split method this time and labeled the three lines with the ROC AUC estimators.

```
10-fold cross validation:

ROC AUC: 0.99 (+/- 0.01) [Logistic regression]
ROC AUC: 0.95 (+/- 0.03) [Decision tree]
ROC AUC: 0.99 (+/- 0.01) [kNN]
```




**Discussion of the ROC curve and AUC for the three models**

From cross validation there are almost miniscule differences between logistic regression and KNN. They maximize the ROC AUC with a.99 (+/-0.01 standard deviation). However, when we were utilizing train_test_split, we got different results for the ROC AUC, and saw that Logistic regression performed a little better than the Knn and the graph shows the difference. It is the best overall compared to KNN and Decision Tree according to the graph with the AUC that is closest to a perfect 1. Based on the above analysis, we came to the conclusion that Logistic Regression would be the best classifier.