EMORY UNIVERSITY

Introduction to Business Data Analytics

Homework #3B

Team 9

Xiao Han, Jent Lapalm, Aoran Wang, Yushan Yang

Part A: Predictive Modeling with Decision Tree, k-NN, and Logistic Regression

In this report, we continue our analysis of the Wisconsin Breast Cancer dataset, which describes cellular features from fine needle aspirate samples of breast mass. Our goal was to train, optimize, and visualize models that correctly classify samples as *benign* or *malignant*. We used the Decision Tree, k-NN, and Logistic Regression modeling techniques. As part of the model-building process, we refined our models using the *nested cross-validation* technique.

Nested Cross-Validation

For readers with a non-technical background, here is a general description of *nested cross-validation*. When training predictive models, the model metaphorically 'memorizes' the training data. However, this can lead to poor predictions on new data, which often does not match its training data. Thus, in order to lessen the impact of training data idiosyncrasies, we refine our model using multiple, different sets of randomly selected training and testing data. This is called the *inner loop*. We perform a similar process for a second loop, called the *outer loop*, which returns model performance for multiple train/test datasets, giving us a wider view of how well our model performs (eg. "our model is 95% accurate within these bounds).

Measuring Performance

There is no single metric to describe how well a model performs. Instead, as a data scientist, it is important to choose performance metrics on a case-by-case basis. Because we are predicting whether someone has cancer, the performance metric we choose has very serious outcomes. Specifically, we want to maximize correctly predicting cancer (ie, true positives) and minimize incorrect non-cancer predictions (ie, false negatives). One additional consideration is that our data has an imbalance of benign and malignant cases, which we would also like to account for. For these reasons, our team selected 'weighted f1-measure' as the performance indicator we are optimizing. For the non-technical reader, this metric takes both true positives and false negatives into account, *and* is weighted to address discrepancies in the number of each case we have.

Decision Tree: Process, Optimal Parameters, Model Performance

Process:

In order to optimize our Decision Tree parameters, we used the 'GridSearchCV' tool. This function uses the aforementioned *inner loop* cross-validation to test which parameters perform best. As discussed previously, we were optimizing the *weighted f1-measure*.

For decision trees, we want to use the most predictive splitting criterion (ie. how our tree chooses to 'branch'), how 'deep' our tree should go, how few 'samples' or data points can be allowed in a single tree leaf, and how few samples are needed for a node to split into two more leafs. Tuning depth, leaf-samples, and split-samples helps prevent the model from memorizing and overfitting the training data. Therefore, this optimization returns a model which can fit not only the existing data but new data.

Finally, we used the *outer loop* cross-validation (discussed above) to generalize our model's performance.
[1] https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

Code:

```
inner_cv = KFold(n_splits = 5, shuffle=True)
outer_cv = KFold(n_splits = 5, shuffle=True)

from sklearn.tree import DecisionTreeClassifier
# Choosing depth of the tree AND splitting criterion AND min_samples_leaf AND min_samples_split

gs = GridSearchCV(estimator = DecisionTreeClassifier(random_state=42),
                  param_grid = [{'criterion' : ['gini', 'entropy'],
                                 'max_depth' : [2,3,4,5,6,7,8,9,None],
                                 'min_samples_leaf' : [1,2,3,4,5,6,7,8,9],
                                 'min_samples_split' : [2,3,4,5,6,7,8,9]
                                  }],
                  scoring = 'f1_weighted',
                  cv = inner_cv,
                  n_jobs = -1)

gs = gs.fit(X,y)
print("Parameter Tuning Decision Tree")
print("Non-nested CV f1 Score: ", gs.best_score_)
# Parameter setting that gave the best results on the hold out data.
print("Optimal Parameter: ", gs.best_params_)
# Estimator that was chosen by the search, i.e. estimator which gave highest score
print("Optimal Estimator: ", gs.best_estimator_)
nested_score_gs = cross_val_score(gs, X = X, y = y, cv = outer_cv)
print("Nested CV f1 Score: ",nested_score_gs.mean(), " +/- ", nested_score_gs.std())
```

```
Parameter Tuning Decision Tree
Non-nested CV f1 Score:  0.9576841091924265
Optimal Parameter:  {'criterion': 'gini', 'max_depth': 5, 'min_samples_leaf': 6, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeClassifier(max_depth=5, min_samples_leaf=6, random_state=42)
Nested CV f1 Score:  0.9400724160975946  +/-  0.02038080911313938
```

Results:

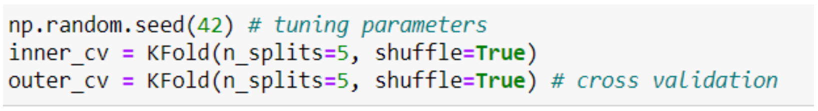
The optimal parameter is the best-performing combination of the previously discussed modeling parameters, while the optimal estimator is the specific decision tree function in python with all of its parameters (including defaults) set as values. The non-nested CV f1 is the exact f1-measure from the optimal model. Finally, the nested CV f1 is the f1-measure after nested cross-validation. The nested CV f1 is slightly lower than the non-nested CV, as it averages performance in order to reduce generalization error.

k-NN: Process, Optimal Parameters, Model Performance

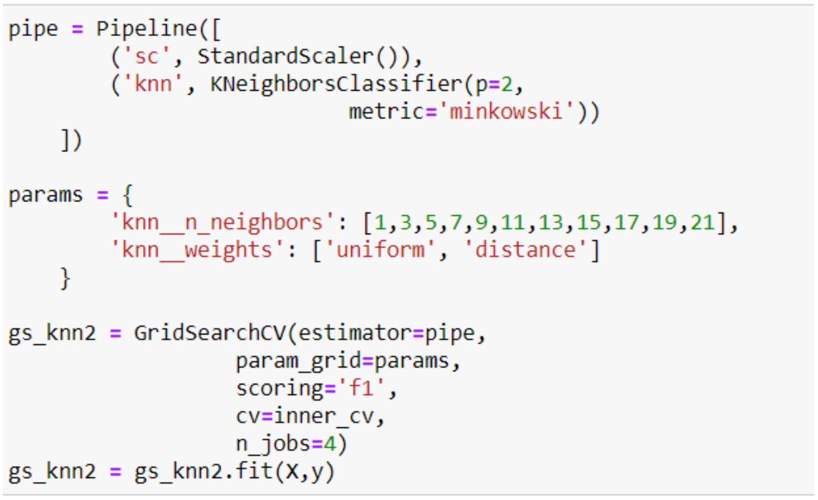
Process:

For the k-NN model, we chose to optimize the number of neighbors and how to weigh the distances between neighbors. For the *k neighbors* parameter, the higher the number of neighbors we choose, the smoother the k-NN curve, and the less complicated our k-NN model will be. However, this might result in overfitting, so we decided to restrict the number of potential neighbors to a range of odd numbers between one and twenty-one. We choose all odd numbers here since it would be intuitively easier for the model to choose the nearest neighbor (ie, when an even number votes, there can be ties, but when an odd number votes, there are no ties). For the *weights* parameter, 'uniform' means all points in each neighborhood are weighted equally, and 'distance' means that we weight points by the inverse of their distance from the instance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

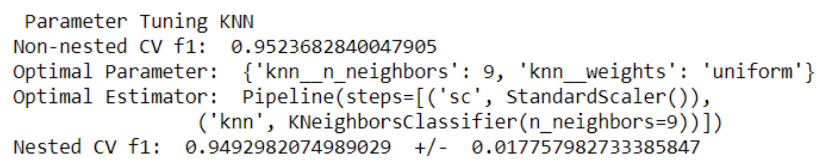
Code:



```
np.random.seed(42) # tuning parameters
inner_cv = KFold(n_splits=5, shuffle=True)
outer_cv = KFold(n_splits=5, shuffle=True) # cross validation
```



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

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)
```



```
 Parameter Tuning KNN
Non-nested CV f1:  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:  0.9492982074989029  +/-  0.017757982733385847
```

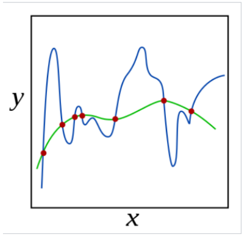
Results:

As a result, we found that the best number of neighbors in our range is 9, and the best weights metric is uniform. In this case, our model yields a generalized f1-measure in a range of 0.9492+/- 0.1776. Like discussed previously, using nested cross-validation effectively prevents the model from memorizing the training dataset while tuning parameters and returns a range for generalized performance. For readers keeping track, it appears our k-NN model performed *better* than the decision tree. However, there is one additional model to consider.

Logistic Regression: Process, Optimal Parameters, Model Performance

Process:

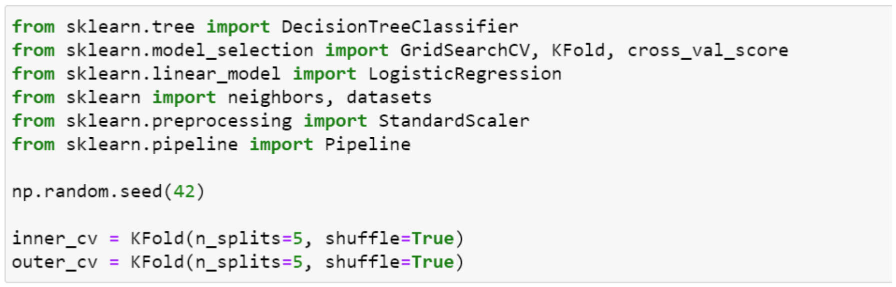
For our third model, logistic regression, we decided to optimize two parameters: *C* and *penalty*. *C* is the inverse of regularization strength. For the non-technical reader, this parameter aims to reduce overfitting to training data by providing a penalty. This may be easier to 'see' than read, so please consider the following graph showing the effects of regularization; while both curves match the red data points, the green curve has been regularized and thereby, hopefully better generalizes to new data points.



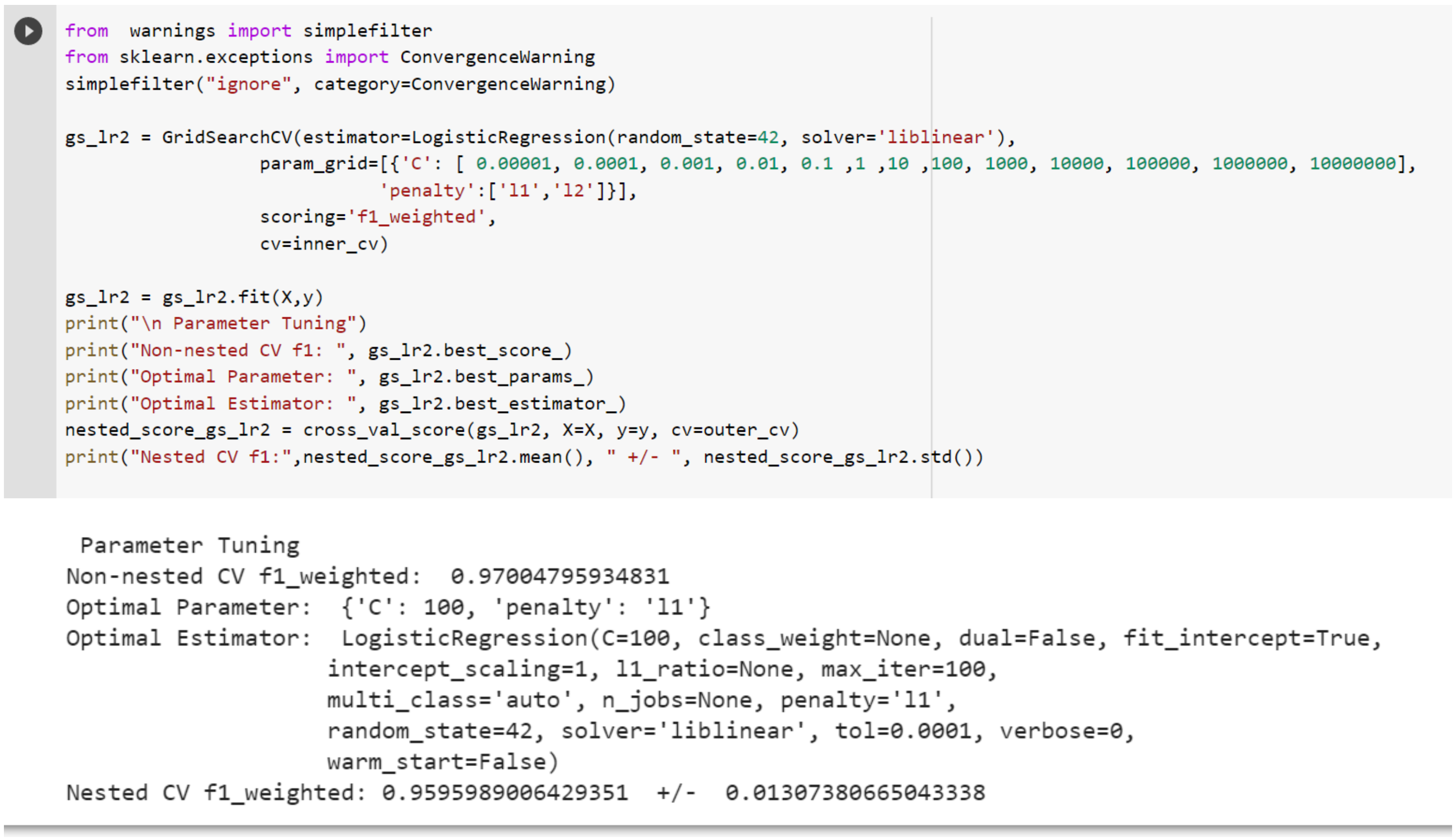
[2] https://en.wikipedia.org/wiki/Regularization_(mathematics)

We tested a wide range of *C* values, from 0.00001 to 10000000. Logistic regression has an additional regularization parameter, *penalty*. For the non-technical reader, l1 and l2 penalties, (ie, lasso and ridge regressions) differ the most in terms of feature selection. The l1 penalty will actually 'zero-out' less predictive features. It is a great tool for datasets with many predictive features (like ours).

Code:



```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV, KFold, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn import neighbors, datasets
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

np.random.seed(42)

inner_cv = KFold(n_splits=5, shuffle=True)
outer_cv = KFold(n_splits=5, shuffle=True)
```



```
from  warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
simplefilter("ignore", category=ConvergenceWarning)

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_weighted',
                  cv=inner_cv)

gs_lr2 = gs_lr2.fit(X,y)
print("\n Parameter Tuning")
print("Non-nested CV f1: ", 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:",nested_score_gs_lr2.mean(), " +/- ", nested_score_gs_lr2.std())
```

```
 Parameter Tuning
Non-nested CV f1_weighted:  0.97004795934831
Optimal Parameter:  {'C': 100, 'penalty': 'l1'}
Optimal Estimator:  LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l1',
                   random_state=42, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
Nested CV f1_weighted: 0.9595989006429351  +/-  0.013073806650433338
```

Results:

Looking at these results, we see that for the optimal logistic regression model, *C* is 100 and *penalty* is l1. As noted earlier, l1 often shines on datasets with many features such as ours. The non-nested CV weighted f1-measure was 0.97 and nested CV weighted f1-measure was 0.96. These are the best results of the three models we used.

Model Comparison and Selection

As discussed earlier, because we are predicting occurrences of cancer, we wanted to take a conservative approach that maximized correct cancer predictions and minimized incorrect non-cancer predictions. The weighted f1-measure gives us the best evaluation of how well our models meet *both* goals. From this metric alone, we see the logic regression provided the best performance. We would choose Logistic Regression were we to deploy this model in a medical setting.

Why was this the case? One challenge with the data was its very high number of features. In data science there are many tools to address this, however, within the scope of this project, only the logistic regression model explicitly included feature selection via the l1 penalty. While decision trees implicitly select features via information gain, they do not remove features by default. k-NN models are also known to suffer from this 'curse of dimensionality', though again there are techniques beyond the scope of this project that can address this.

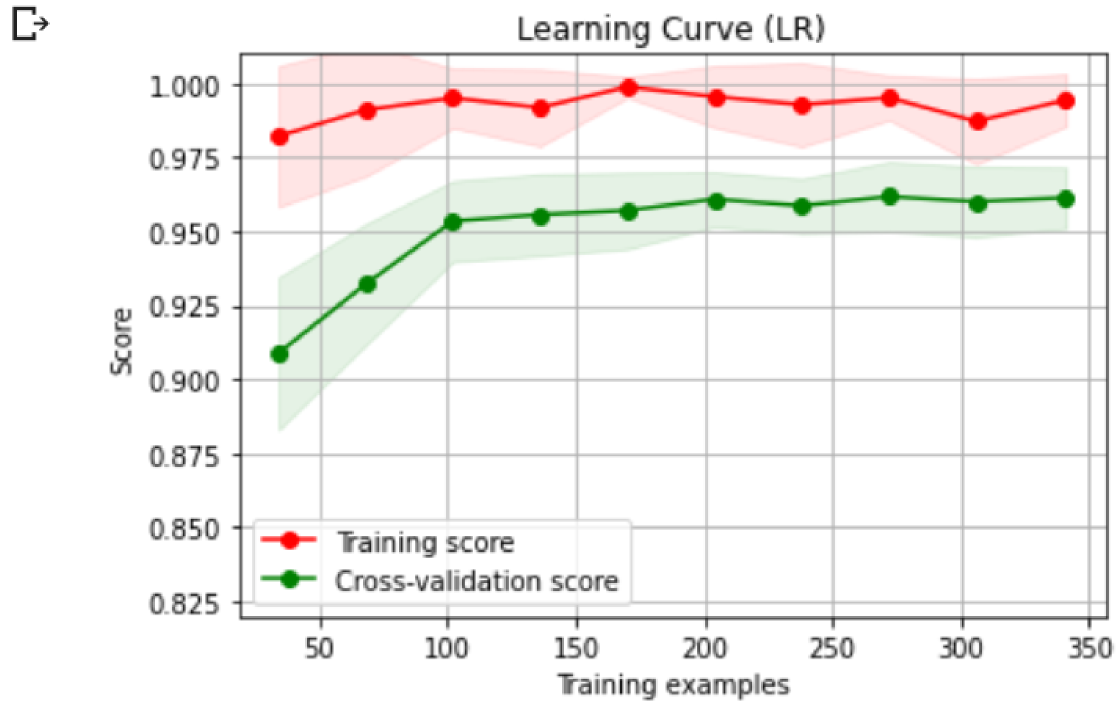
Were our team to revisit this project in the future, we would like to explore using attribute transformation and PCA in conjunction with our earlier modeling techniques to see if these results persist. That the k-NN algorithm performed *nearly* as well as logistic regression is compelling.

Part B: Logistic Regression Learning Curve and Process

Process:

Learning curves are a way to assess the performance improvement of adding more data. These are especially useful when there is a cost associated with gathering more data. For example, it may be very expensive for the University of Wisconsin to gather more samples for this dataset; the learning curve helps quantify the improvements in model performance as a result of this additional data.

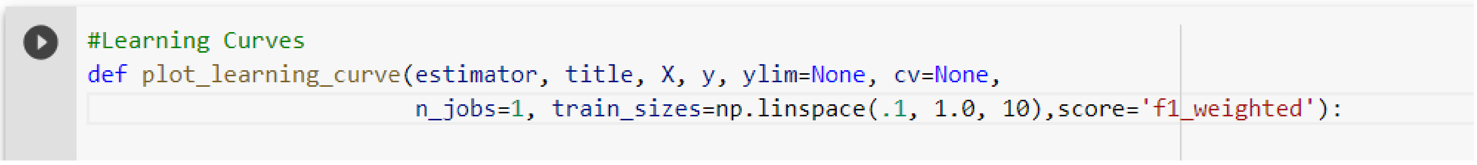
The process for plotting a learning curve is fairly self-evident from the graph: determine cross-validated train and test scores from a certain size of test data and plot these results. The red and green highlighted areas of the graph represent one standard deviation above and below the mean, which represent roughly 68% of generalized model performance. We are able to make this generalization because of cross-validation, which was discussed earlier in this report.



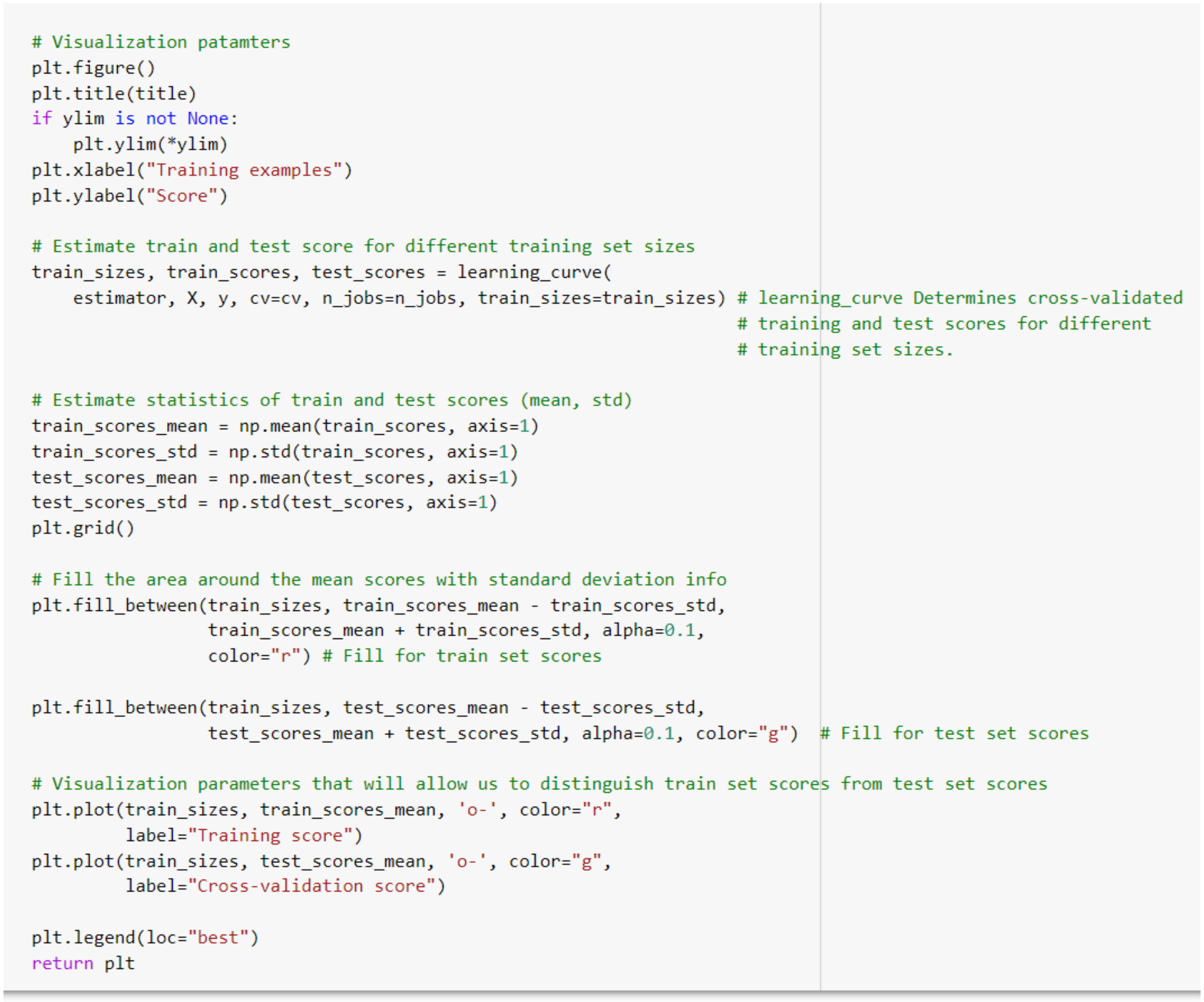
Results:

From the above learning curve, we can see the training score (weighted f1-measure) increased overall as training examples increased. For the training score, it was increased as training examples increased. The cross-validation score was increasing then decreasing. This means that with more and more samples, the progression of learning is initially fast, but after more training examples made the function too complex, the decreasing cross-validation score indicated that overfitting was possibly occurring; this occured when training examples were between 200 and 250. However, as training examples continued increasing, there was no trend of overfitting because the cross-validation score was stable. From these results, it appears adding more data after a certain point, (ie. the 200th example) does not have a large, positive impact on model performance.

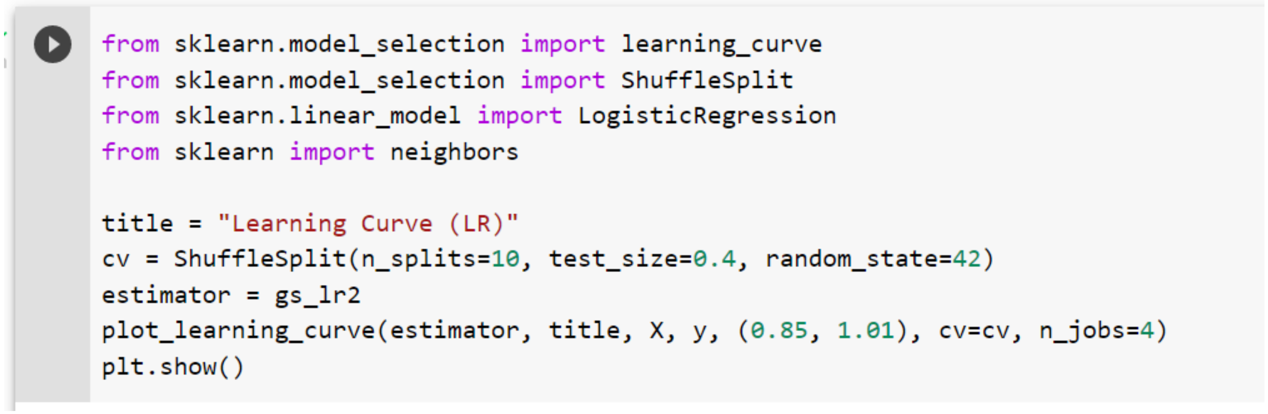
Code:



```
#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),score='f1_weighted'):
```



```
# Visualization patamters
plt.figure()
plt.title(title)
if ylim is not None:
    plt.ylim(*ylim)
plt.xlabel("Training examples")
plt.ylabel("Score")

# Estimate train and test score 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.

# Estimate statistics of train and test scores (mean, std)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)
plt.grid()

# Fill the area around the mean scores with standard deviation info
plt.fill_between(train_sizes, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.1,
                 color="r") # Fill for train set scores

plt.fill_between(train_sizes, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.1, color="g")  # Fill for test set scores

# 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")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g",
         label="Cross-validation score")

plt.legend(loc="best")
return plt
```



```
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from sklearn.linear_model import LogisticRegression
from sklearn import neighbors

title = "Learning Curve (LR)"
cv = ShuffleSplit(n_splits=10, test_size=0.4, random_state=42)
estimator = gs_lr2
plot_learning_curve(estimator, title, X, y, (0.85, 1.01), cv=cv, n_jobs=4)
plt.show()
```

Part C: Decision Tree Fitting Graph and Process

Process:

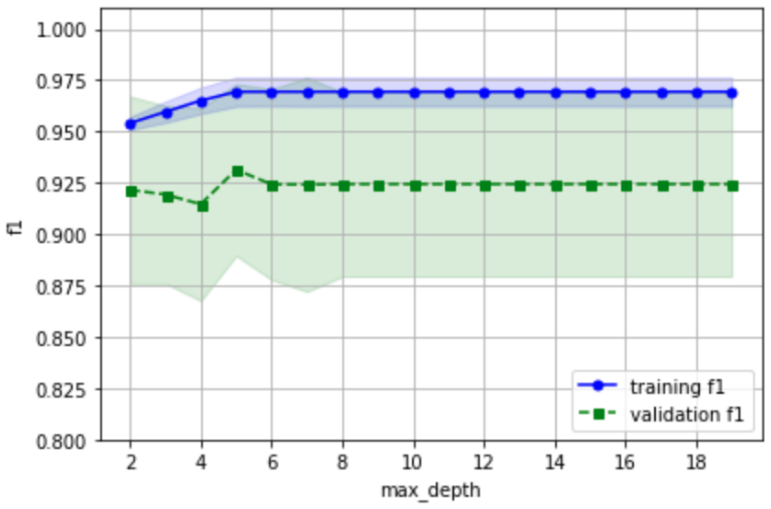
A fitting graph is useful for evaluating how well decision trees perform given a particular depth. When training performance increases as test performance decreases, the graph serves as a visual warning that the model may be overfitting training data.

In-sample and out-of-sample performance are generated based on training and test sets. The fitting graph of the decision tree model plots the f1-measure against the depths of the model. The steps of drawing a fitting graph of in-sample performance are:

1. Get the in-sample performance: the optimal model generated in part a) which predicts the diagnosis based on the training set.
2. Obtain the f1-measure: as discussed in part a), the f1-measure is our selected scoring method.
3. Set the parameter: the parameter name and range are set as max_depth to generate performance for the depth of the decision tree model from 2 to 20.
4. Plot the f1-measure against depths of the decision tree model.

The steps of drawing a fitting graph of out-of-sample performance are similar:

1. Get the out-of-sample performance and the f1-measure: the optimal model generated in part a) predicts the diagnosis using the test set.
2. Repeat step 2~4 of the in-sample graphing process, which draws the fitting graph.



<u>Results</u>:

The blue line which represents the in-sample performance is above the green line which represents the out-of-sample performance. Generally, the out-of-sample performance will be worse than the in-sample performance, because the model tends to overfit the training model, resulting in a relatively better in-sample performance. The training f1-measure keeps increasing until it reaches a maximum when the depth of the decision tree model grows more than 5, after which tree depth does not increase training performance. Consistent with the optimal model generated from part a), the validation f1-measure is largest when the depth of the decision tree model is 5. The out-of-sample performance is optimal in this model. The green line decreases, then fluctuates, indicating that more tree depth does not necessarily result in better out-of-sample performance.

Code:

```python
from sklearn.model_selection import validation_curve
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

np.random.seed(42)

max_depth = range(2,20,1)
param_name = "max_depth"
param_range = max_depth

DTmodel = tree.DecisionTreeClassifier(criterion = 'gini',
                                      min_samples_leaf = 6,
                                      min_samples_split = 2,
                                      random_state = 42
                                     )

# Determine training and test scores for varying parameter values.
train_scores, test_scores = validation_curve(
                DTmodel,
                X = X_train,
                y = y_train,
                param_name = param_name,
                param_range = param_range,
                cv = 10,     #10-fold cross-validation
                scoring = 'f1_weighted',
                n_jobs = -1
) # Number of CPU cores used when parallelizing over classes if multi_class='ovr'".
# This parameter is ignored when the ``solver``is set to 'liblinear'
# regardless of whether 'multi_class' is specified or not.
# If given a value of -1, all cores are used.
```

```python
# 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)


# 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')

# 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 accuracy 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='validation f1')

# 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')
```

```python
# Grid and Axes Titles
plt.grid()
#plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('max_depth')
plt.ylabel('f1')
plt.ylim([0.8, 1.01]) # y limits in the plot

from matplotlib.ticker import MaxNLocator
plt.gca().xaxis.set_major_locator(MaxNLocator(integer=True))

plt.tight_layout()
# plt.savefig('Fitting_graph_LR.png', dpi=300)
plt.show()           # Display the figure
```

Part D: ROC Curve for k-NN, Decision Tree, and Logistic Regression, Process

Process:

The ROC Curve (receiver operating characteristic) looks at a model's true positive rate as a function of its false positive rate. The diagonal line on the graph represents random guessing, which we can ballpark our models against. For the non-technical reader, please consider the North-West portion of the graph to be the 'best' outcomes, with a caveat: depending on the specific case, we may want to enforce a particular true or false positive rate. For our models, true positives (correctly identifying cancer) are *much* more important than false positives, which, while distressing, would be caught by a follow-up test and are not life-threatening. Therefore, we should limit our interest to the northern sector of the graph, which fortunately, all our models occupy.

In order to build the ROC graph, we took our optimized models and predicted the *probability* of a data point being malignant or benign. Next, we calculate the true and false positive rate given by probability thresholds between 0 and 1. For example, we could set a threshold of 0.6 meaning all predictions with a lower probability are considered negative and all predictions with a higher probability considered positive. These rates correspond to *y* and *x*, respectively, in the coordinate plane, and were graphed to create the ROC curve.

While building this graph, we took care to normalize the data for our k-NN model, and use raw data for the decision tree and logistic regression, as we did when optimizing them.

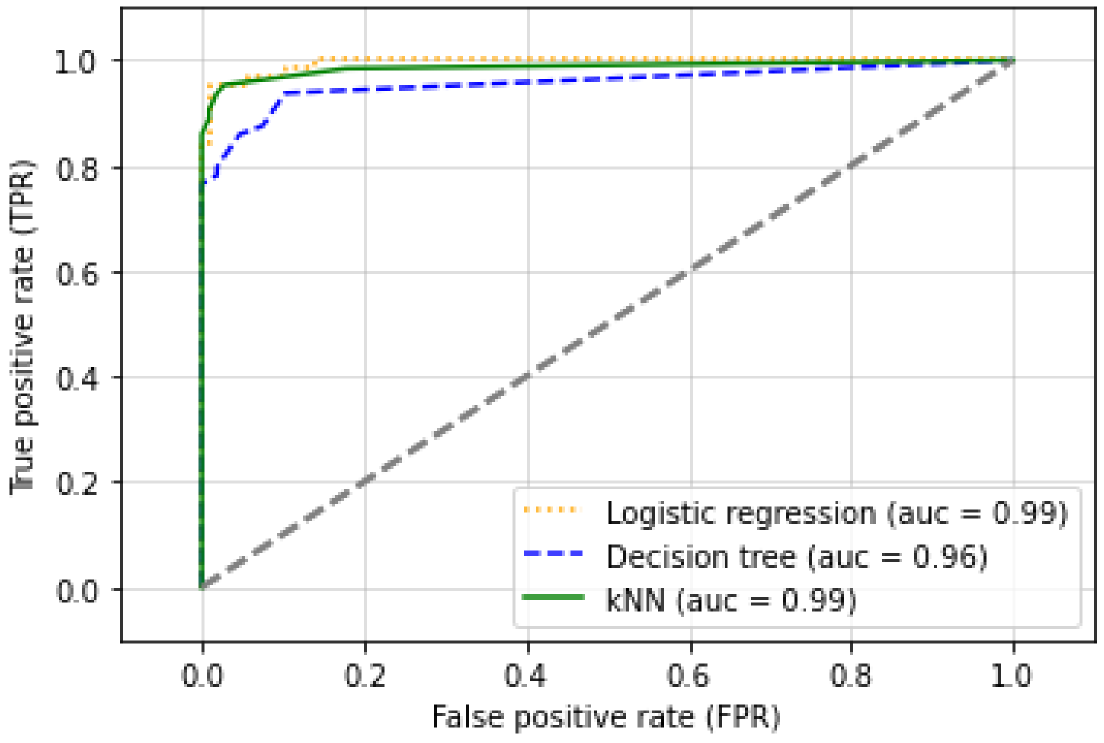
Results:

For the non-technical reader: as mentioned earlier, the North-West portion of the graph represents the best model. Logistic regression is *slightly* ahead of the other models.

Diving into more technical metrics, it is important to note the AUC of each model. AUC represents the area under the ROC curve, and is a measure of how well a model distinguishes between positive and negative classes. An AUC of 1 is a perfect model, so we were very pleased to see such high AUC scores for all of our models.

Choosing a classifier:

Looking only at the ROC curve and AUC values, one might have trouble deciding between logistic regression and k-NN. However, earlier in this report we noted logistic regression's superior performance on the weighted f1-measure, and discussed why this metric best measured our model's successes. Given the equal performance on AUC, because logistic regression performs better on our other metrics, we feel confident in our previous recommendation that logistic regression is the best model for this data.



Code:

```python
#normalizing data because attributes have different ranges
sc = StandardScaler()
sc.fit(X_train)

X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

######################################### 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',
                              random_state=42,
                              min_samples_leaf=6,
                              min_samples_split=2)

# kNN Classifier
clf3 = KNeighborsClassifier(n_neighbors=9,
                            p=2,
                            metric='minkowski',
                            weights='uniform')

# Label the classifiers
clf_labels1 = ['Logistic regression', 'Decision tree']
all_clf1 = [clf1, clf2]
clf_labels2 = ['kNN']
all_clf2 = [clf3]
###################################### Visualization #######################################
colors1 = [ 'orange', 'blue', 'green']      # Colors for visualization
linestyles1 = [':', '--', '-.', '-']        # Line styles for visualization
#ROC curve for decision tree, Logit
for clf, label, clr, ls in zip(all_clf1,
               clf_labels1, colors1, linestyles1):

    # 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='--',
         color='gray',
         linewidth=2)

#ROC curve for k-NN
colors2 = ['green']      # Colors for visualization
linestyles2 = ['-']        # Line styles for visualization
for clf, label, clr, ls in zip(all_clf2,
               clf_labels2, colors2, linestyles2):

    # Assuming the label of the positive class is 1 and data is normalized
    y_pred = clf.fit(X_train_std,
                     y_train).predict_proba(X_test_std)[:, 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)
```