Machine Learning

Debugging and Evaluation

Part One

Dr. Sherif Saad

Learning Objectives

Understand best practices for evaluating machine learning algorithms

Learn how to build good training sets

Diagnose the common problems of machine learning algorithms

Understanding and applying different performance metrics

Outlines

- PART ONE
  - Introduction to Machine Learning Evaluation
  - Prepare for Model Evaluation
  - Overfitting and Underfitting
- PART TWO
  - Debugging Machine Learning Algorithms
  - Performance Metrics

Underfitting | Overfitting

Overfitting in this case the machine learning model performs well on training data but does not generalize well to unseen data (test data).

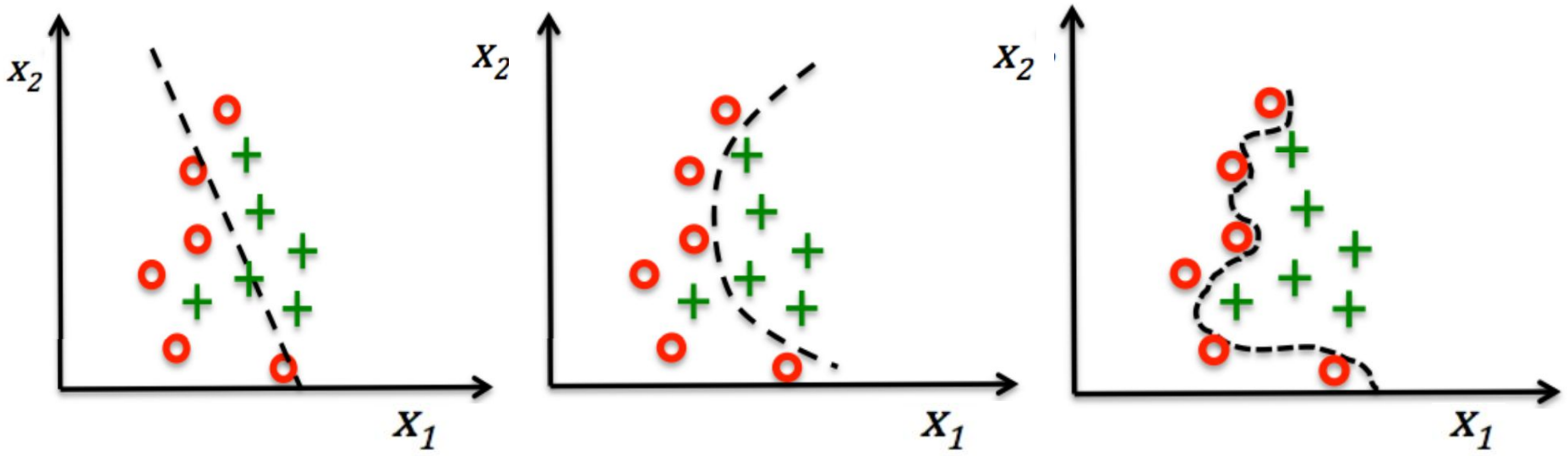
If a model suffers from overfitting, we also say that the model has a high variance. This is usually happen when we have too many parameters (features) that lead to a model that is too complex given the underlying data.

Underfitting on the other hand means our model is not complex enough (low learning ratio) to capture the patterns and the structure of the data. Will perform poorly on unseen data. Also known as high bias.

Usually we apply regularization methods to find a balance between overfitting and underfitting.

Underfitting | Overfitting

You have three models (black dashed lines) which one overfit the data, underfit the data and which one is a good balance between overfitting and underfitting



Debugging ML Algorithms

How could we detect if our ML model suffers from overfitting (high variance) or underfitting (high bias)?

How could we detect other potential issues that could affect the performance of our ML model?

Two simple diagnostic methods, namely, learning curves and validation curves.

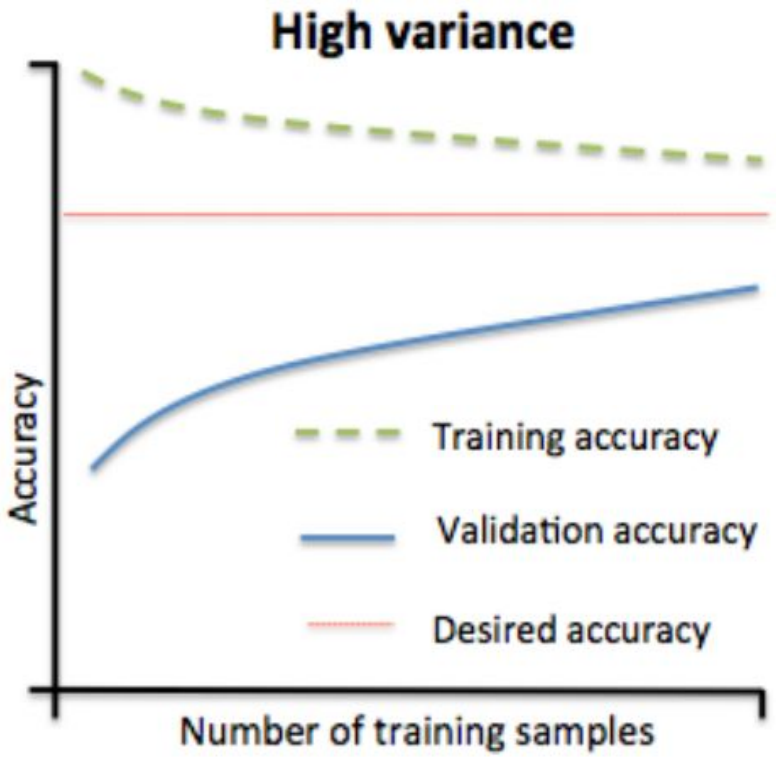
We can use learning curves to detect if our ML model has a problem with overfitting or underfitting.

Detect Overfitting With Learning Curve

The large gap between the training and cross-validation accuracy indicate overfitting.

Collect more training data or reduce the complexity of the model.

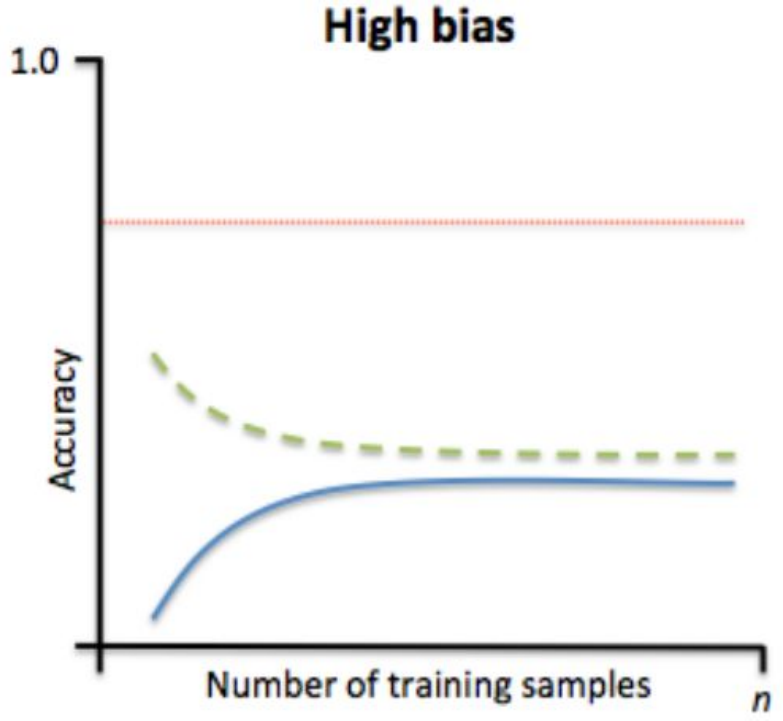
Decrease the number of features via feature selection or feature extraction.



Detect Underfitting With Learning Curve

Has both low training and cross-validation accuracy, which indicates that it underfits the training data.

Increase the number of parameters of the model, for example, by collecting or constructing additional features



Debugging ML Algorithm with Learning Curve: Example

Breast Cancer Wisconsin Dataset

- Has 569 samples
- Each sample has 30 features
- Binary classification problem malignant tumors M and benign tumors B

Breast Cancer Wisconsin Dataset

```
# pandas to load the dataset
import pandas as pd

# we use this module to encode the labels from B and M to 0 and 1
from sklearn.preprocessing import LabelEncoder

# we use this module to split the data into training and testing
from sklearn.model_selection import train_test_split

# this the ML module we are going to use (evaluate)
from sklearn.linear_model import LogisticRegression

# we use this module to scale or normalize the feature valued between 0 and 1
from sklearn.preprocessing import StandardScaler

# we use this module to do feature reduction
from sklearn.decomposition import PCA

# we build a pipeline to apply preprocessing, encoding, feature extraction, selection, and model evaluation
from sklearn.pipeline import Pipeline
```

Breast Cancer Wisconsin Dataset

```
# we use this module to perfrom cross validation
from sklearn.model_selection import cross_val_score

# we use this module to debug the ML model using learning curve
from sklearn.model_selection import learning_curve

# we use numpy for array manipulation
import numpy as np
# we use this module for plotting the learning curve
import matplotlib.pyplot as plt
```

Loading and Exploring the Data

```
# the data set has 568 samples (case) some of them have malignant tumors M and some have benign tumors
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
                 header=None)

# print the number of samples in the dataset
print(len(df))

# print the number of features in the dataset
print(len(df.columns.values))


# print the name of features in the dataset
print(df.columns.values)

# print the number of features in the dataset
print(df.head(5))
```

Splitting the data into Training and Testing

```
# we select the samples data without the labels (class)
X = df.loc[:, 2:].values

# we select the labels column and encode it into binary values where the malignant tumors are now represented as class 1
# and the benign tumors are represented as class 0

y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)

# divide the dataset into a separate training dataset (80 percent of the data) and
# a separate test dataset (20 percent of the data)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)
```

Initialize a Scikit-Learn Pipeline

```
# learning algorithms require input features on the same scale for optimal performance
# pipeline is an abstract notion, it is not some existing ml algorithm.
# Often in ML tasks you need to perform sequence of different transformations
# (find set of features, generate new features, select only some good features)
# of raw dataset before applying final estimator.
bcw_pipeline = Pipeline([('scl', StandardScaler()), ('pca', PCA(n_components=2)),
                         ('clf', LogisticRegression(random_state=1))])

bcw_pipeline.fit(X_train, y_train)

print('Test Accuracy: %.3f' % bcw_pipeline.score(X_test, y_test))


scores = cross_val_score(estimator=bcw_pipeline, X=X_train, y=y_train, cv=10, n_jobs=1)

print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

Constructing Learning Curve

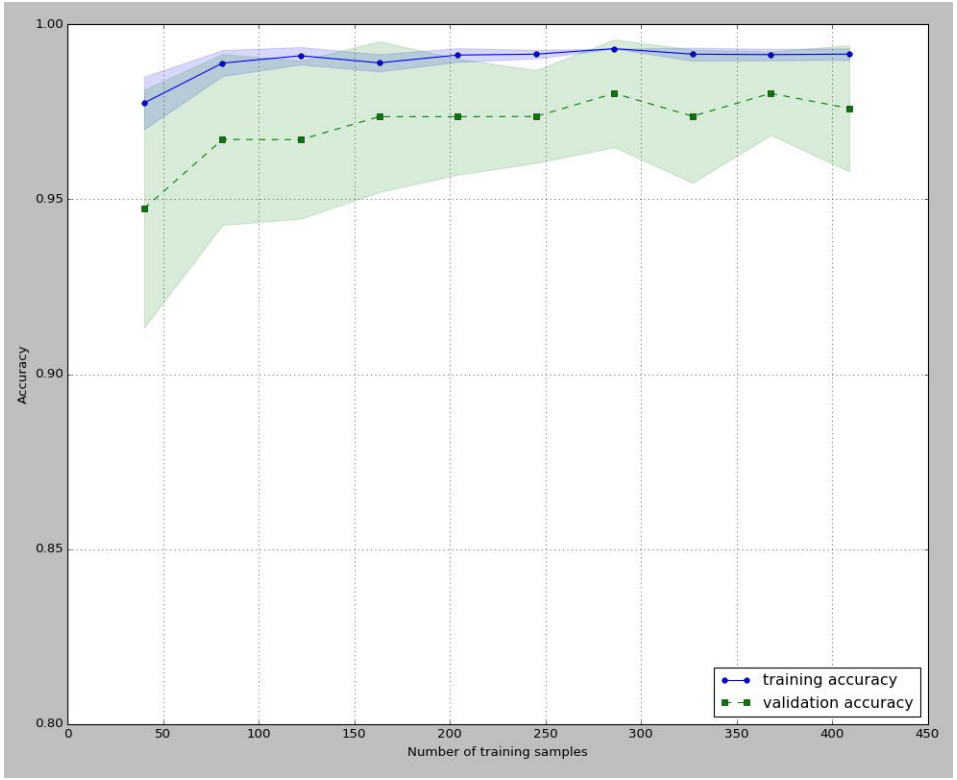
```
# we use learning curve for plotting the training and test accuracies as functions of the sample size
lr_curve_pipeline = Pipeline([('scl', StandardScaler()),
                              ('clf', LogisticRegression(penalty='l2', random_state=0))
                              ])
train_sizes, train_scores, test_scores = learning_curve(estimator=lr_curve_pipeline, X=X_train, y=y_train,
                                                        train_sizes=np.linspace(0.1, 1.0, 10), cv=10, n_jobs=1)

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)
```

Plotting Learning Curve

```
plt.plot(train_sizes, train_mean, color='blue', marker='o', markersize=5, label='training accuracy')
plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std, alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean, color='green', linestyle='--', marker='s', markersize=5, label='validation accuracy')
plt.fill_between(train_sizes, test_mean + test_std, test_mean - test_std, alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.0])
plt.show()
```

Learning Curve

Do you think we have overfitting or underfitting based on the learning curve on the right side?



Addressing Overfitting | Underfitting using validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting.

The idea is we vary the values of the ML model parameters to improve the model performance and address possible overfitting or underfitting

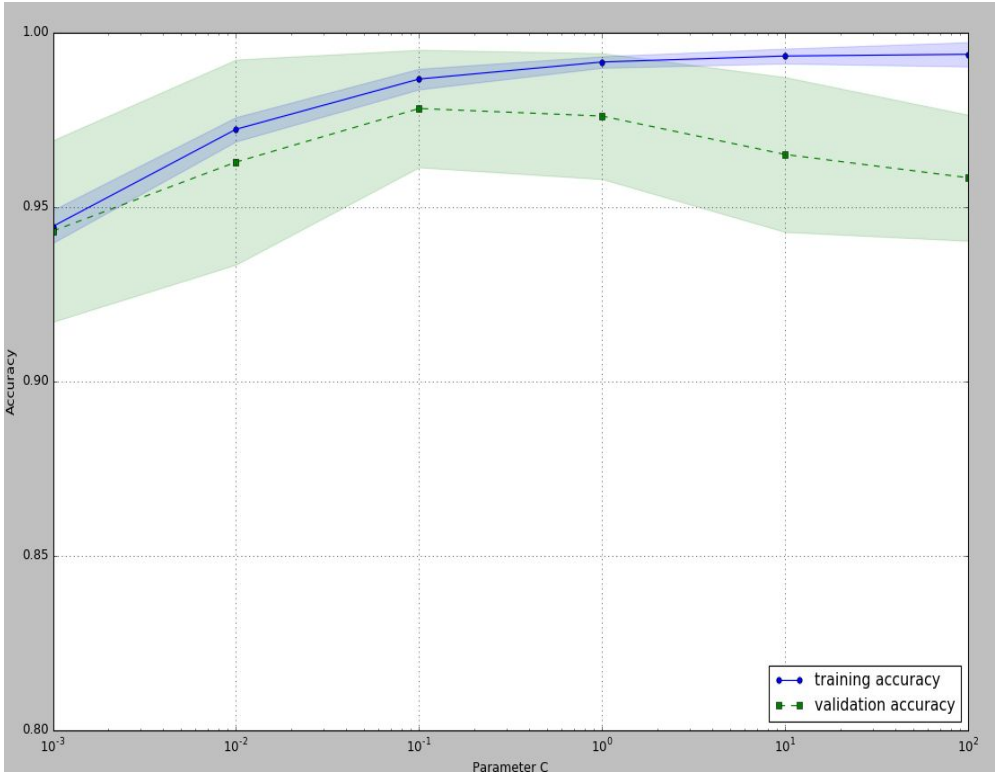
Constructing Validation Curve

```
from sklearn.model_selection import validation_curve

# we vary the values of the model parameters,
# for example, the inverse regularization parameter C in logistic regression
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

train_scores, test_scores = validation_curve( estimator=lr_curve_pipeline, X=X_train, y=y_train,
                                              param_name='clf__C', param_range=param_range, cv=10)

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)

plt.plot(param_range, train_mean, color='blue', marker='o', markersize=5, label='training accuracy')
plt.fill_between(param_range, train_mean + train_std, train_mean - train_std, alpha=0.15, color='blue')
plt.plot(param_range, test_mean, color='green', linestyle='--', marker='s', markersize=5, label='validation accuracy')
plt.fill_between(param_range, test_mean + test_std, test_mean - test_std, alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.8, 1.0])
plt.show()
```

Validation Curve

Given param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

What do you think we have overfitting or underfitting and where based on the validation curve on the right?



Tuning Hyperparameters

In general, every ML algorithm has two types of parameters those that are learned from the training data, for example, the weights in neural network, and the parameters of a learning algorithm that are optimized separately, for example the K in KNN or depth in decision tree.

One method for hyperparameter optimization is called grid search that attempt to find the optimal combination of hyperparameter values.

Grid search is quite simple, it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters and the algorithm evaluates the model performance for each combination of those to obtain the optimal set

Performance Evaluation Metrics

Performance measures are typically specialized to the class of problem you are working with.

Many standard performance measures will give you a score that is meaningful to your problem domain. You may also want a more detailed breakdown of performance, for example, you may want to know

Common Performance Evaluation Metrics

- Classification Accuracy | Logarithmic Loss | AUC - ROC
- Confusion Matrix | Root Mean Squared Error

Classification Accuracy

The most common evaluation metrics is the model accuracy or classifier accuracy. It is also the most misused and misunderstood metric

This is a useful metric to quantify the performance of a model in genera.

It is simply the number of correct predictions made as a ratio of all predictions made.

When the performance metric is meaningful?

- When the number of observations (samples) in each class is close or equal. When the predications and the predictions errors are equally important.

Classification Accuracy

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

# the data set has 568 samples (case) some of them have malignant tumors M and some have benign tumors
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
                 header=None)

# we select the samples data without the labels (class)
X = df.loc[:, 2:].values

# we select the labels column and encode it into binary values where the malignant tumors are now represented as class 1
# and the benign tumors are represented as class 0

Y = df.loc[:, 1].values
le = LabelEncoder()
Y = le.fit_transform(Y)

seed = 7
kfold = KFold(n_splits=10, random_state=seed)
model = LogisticRegression()
scoring = 'accuracy'
results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("Accuracy: mean {0} and std {1}".format(results.mean()*100, results.std()*100))
```

Logarithmic Loss

Log loss or logarithmic loss are used to evaluating the predictions of probabilities of membership to a given class.

It is used when the model outputs a probability for each class, rather than just the most likely class.

Log Loss quantifies the accuracy of a classifier by penalising false classifications. Minimising the Log Loss is basically equivalent to maximising the accuracy of the classifier

Commonly used with multi-class classification problems.

Logarithmic Loss

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$$

where N is the number of observations, M is the number of class labels, *log*log is the natural logarithm, $\boldsymbol{y_{i,j}}$ is 1 if observation *i* is in class *j* and 0 otherwise, and $\boldsymbol{p_{i,j}}$ is the predicted probability that observation i is in class j.

A perfect classifier would have a Log Loss of precisely zero. Less ideal classifiers have progressively larger values of Log Loss.

Logarithmic Loss

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

# the data set has 568 samples (case) some of them have malignant tumors M and some have benign tumors
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
                 header=None)

# we select the samples data without the labels (class)
X = df.loc[:, 2:].values

# we select the labels column and encode it into binary values where the malignant tumors are now represented as class 1
# and the benign tumors are represented as class 0

Y = df.loc[:, 1].values
le = LabelEncoder()
Y = le.fit_transform(Y)

seed = 7
kfold = KFold(n_splits=10, random_state=seed)
model = LogisticRegression()

scoring = 'neg_log_loss'
results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
# scikit-learn implements metrics this way so that larger is better (i.e., to maximize score).
# scikit-learn unified scoring API always maximizes the score, so scores which need to be minimized
# are negated in order for the unified scoring API to work correctly.
print("Log Loss: mean {0} and std {1}".format(results.mean()*100, results.std()*100))
```

Area Under ROC Curve (AUC - ROC)

The Area Under the Receiver Operating Characteristic curve. Is an evaluation metric for binary classifiers.

Measure the ability of the model to discriminate between positive and negative classes.

An area of 1.0 represents a model that made all predictions perfectly. On the other hand an area of 0.5 represents a random model.

We can think of ROC as the sensitivity and the specificity of the model.

Area Under ROC Curve (AUC - ROC)

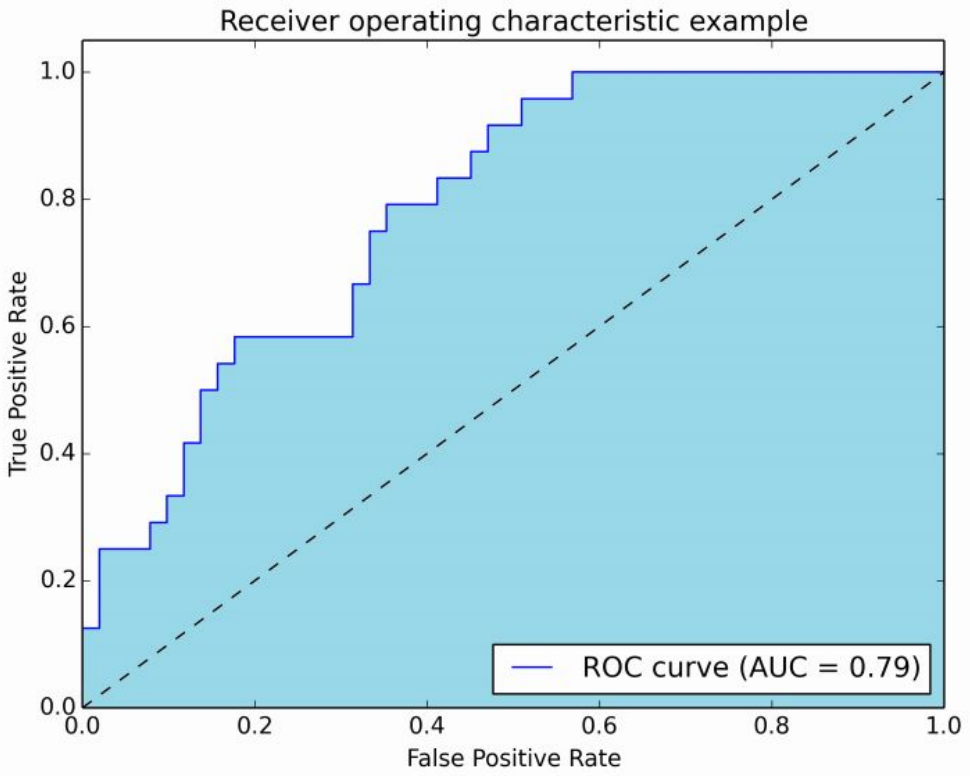
Sensitivity is the true positive rate also called the recall or hit rate. It is the number instances from the positive (first) class that actually predicted correctly.

Sensitivity is also known as True Positive

Specificity is also called the true negative rate or fall-out. Is the number of instances from the negative class (second) class that were actually predicted correctly.

Specificity is also known as False Positive

Area Under ROC Curve (AUC - ROC)



Area Under ROC Curve (AUC - ROC)

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

# the data set has 568 samples (case) some of them have malignant tumors M and some have benign tumors
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
                 header=None)

# we select the samples data without the labels (class)
X = df.loc[:, 2:].values

# we select the labels column and encode it into binary values where the malignant tumors are now represented as class 1
# and the benign tumors are represented as class 0

Y = df.loc[:, 1].values
le = LabelEncoder()
Y = le.fit_transform(Y)

seed = 7
kfold = KFold(n_splits=10, random_state=seed)
model = LogisticRegression()

scoring = 'roc_auc'
results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)
print("AUC-ROC: mean {0} and std {1}".format(results.mean()*100, results.std()*100))
```

Confusion Matrix

An N X N matrix, where N is the number of classes being predicted. In binary classification problems N = 2 and the confusion matrix is a 2 x 2 matrix.

		Predicted class	
		Class 1	Class 0
Actual class	Class 1	10 true positives (TP)	2 false negatives (FN)
	Class 0	3 false positives (FP)	35 true negatives (TN)

Confusion Matrix

To understand the confusion matrix;

- **True Negative:** we correctly predict that the class is negative. the proportion of negative cases that were correctly identified.
- **True Positive:** we correctly predict that the class is positive. the proportion of positive cases that were correctly identified.
- **False Negative:** we incorrectly predict that the class is negative. the proportion of positive cases that were incorrectly identified as negative
- **False Positive:** we incorrectly predict that the class is positive. the proportion of negative cases that were incorrectly identified as positive

Confusion Matrix

True Positive Rate: ***TP/(TP+FN)***. The proportion of positive data points that are correctly considered as positive, with respect to all positive data points. In other words, the higher TPR, the fewer positive data points we will miss.

False Positive Rate: ***FP/(FP+TN)***. the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points. In other words, the higher FPR, the more negative data points we will misclassified.

Confusion Matrix

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# the data set has 568 samples (case) some of them have malignant tumors M and some have benign tumors
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',
                 header=None)

# we select the samples data without the labels (class)
X = df.loc[:, 2:].values

# we select the labels column and encode it into binary values where the malignant tumors are now represented as class 1
# and the benign tumors are represented as class 0

Y = df.loc[:, 1].values
le = LabelEncoder()
Y = le.fit_transform(Y)

seed = 7
kfold = KFold(n_splits=10, random_state=seed)
model = LogisticRegression()

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.20, random_state=1)
model = LogisticRegression()
model.fit(X_train, y_train)
predicted = model.predict(X_test)
matrix = confusion_matrix(y_test, predicted)
print("Confusion Matrix {0}".format(matrix))
```

Root Mean Squared Error

RMSE is the most popular evaluation metric used in regression problems.

The square root of the mean/average of the square of all of the error.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where n is the number of samples in the dataset, y_i is the estimated score of the observation i and $\hat{y}_i$ *is actual score of the observation i*

```
from sklearn.metrics import mean_squared_error
RMSE = mean_squared_error(y, y_pred)**0.5
```

Questions