Predictive model selection and evaluation

First of all, we do all necessary imports and load the Breast Cancer Wisconsin Diagnostic dataset.

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, zero_one_loss
from sklearn.metrics import roc_curve, auc
from sklearn.datasets import load_breast_cancer

from scipy.stats import wilcoxon, friedmanchisquare, rankdata
from Orange.evaluation import compute_CD, graph_ranks

DEFAULT_N_NEIGHBORS = 5

# Random seed. This is needed to make all results reproducible.
seed = 10

# Loading Breast Cancer dataset
breast_cancer = pd.read_csv('data/breast_cancer.csv', index_col=0)
labels = 'diagnosis'
breast_cancer.head()
```

Out[1]:

	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	mean_compactness
sample_id						
842302	17.99	10.38	122.80	1001.0	0.11840	0.27760
842517	20.57	17.77	132.90	1326.0	0.08474	0.07864
84300903	19.69	21.25	130.00	1203.0	0.10960	0.15990
84348301	11.42	20.38	77.58	386.1	0.14250	0.28390
84358402	20.29	14.34	135.10	1297.0	0.10030	0.13280

5 rows × 31 columns

Then, we encode the 'diagnosis' str values as int values.

In [16]:

```
le = LabelEncoder()
breast_cancer[labels] = le.fit_transform(breast_cancer[labels])
```

Then, we write a simple method to train, test and return a kNN classifier, its predicted results, its accuracy score and its 0-1 loss for a dataset.

In [3]:

```
def knn_fit_predict_evaluate(X_train, X_test, y_train, y_test, k=DEFAULT_N_NEIGHBORS):
    knn = KNeighborsClassifier(n_neighbors=k, weights='distance', metric='euclidean')
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    loss = zero_one_loss(y_test, y_pred)
    return knn, y_pred, accuracy, loss
```

To train and test sklearn classifiers, we will need the data as numpy arrays. So, we can extract them using the code below.

In [4]:

```
np.set_printoptions(precision=4, suppress=True)

# Extracting Breast Cancer values without the label column.
X = breast_cancer.drop(labels, axis=1).values
print('X:\n{}\n'.format(X))

# Extracting Breast Cancer labels.
y = breast_cancer[labels].values
print('y:\n{}\n'.format(y))
```

```
X:
[[  17.99     10.38    122.8    ...,    0.2654    0.4601    0.1189]
 [  20.57     17.77    132.9    ...,    0.186     0.275     0.089 ]
 [  19.69     21.25    130.     ...,    0.243     0.3613    0.0876]
 ...,
 [  16.6      28.08    108.3    ...,    0.1418    0.2218    0.0782]
 [  20.6      29.33    140.1    ...,    0.265     0.4087    0.124 ]
 [   7.76     24.54     47.92   ...,    0.        0.2871    0.0704]]

y:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1
 0 1 0 1 1 0 0 0 1 1 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 1 0 0
 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 0 1
 0 1 0 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 0 1 1 0 0
 0 1 0 0 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 1 1 0 0
 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1
 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0
 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0
 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1
 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1 1 1 1 1 1 0]
```

Simple holdout

The simple holdout method consists of splitting the original dataset in two disjoint subsets:

- **train**: which contains a proportion of p objects from the original dataset;
- **test**: which contains a proportion of $1 - p$ objects from the original dataset.

The aforementioned split can be performed with the train_test_split method.

In [5]:

```
# Splitting the original dataset.
# The train subset will contain a proportion of 0.66 objects from the original dataset.
# The test subset will contain a proportion of 0.34 objects from the original dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.34, stratify=y, random_state=seed)

# Training, testing and evaluating a kNN classifier with simple holdout.
knn, y_pred, accuracy, loss = knn_fit_predict_evaluate(X_train, X_test, y_train, y_test)
print('Accuracy: {}'.format(accuracy))
print('0-1 loss: {}'.format(loss))

# Since the accuracy and 0-1 loss are complementary measures, their sum must be 1.0.
print('Accuracy + 0-1 loss = {}'.format(accuracy + loss))
```

```
Accuracy: 0.9226804123711341
0-1 loss: 0.07731958762886593
Accuracy + 0-1 loss = 1.0
```

K-fold cross-validation

K-fold cross-validation consists of splitting the original dataset in k disjoint subsets of approximately equal size. Then, at each iteration, $k - 1$ subsets are used as the training set and the remaining subset is used as the test set. In the end, we can calculate the mean accuracy as the performance measure.

Sklearn provides several different classes to perform k-fold cross-validation. In this example, we use the StratifiedKFold class, since it breaks the original dataset in k stratified disjoint subsets. So, each subset will maintain the same proportion of objects in each class as in the original dataset.

In [6]:

```
splits = 10
skfold = StratifiedKFold(n_splits=splits, random_state=seed)
trained_knns = []
accuracies = []

# skfold.split(X, y) returns an iterator over tuples.
# In each tuple, the first element consists of the indices of examples from the train set.
# The second element consists of the indices of examples from the test set.
for train_idx, test_idx in skfold.split(X, y):
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]
    knn, y_pred, acc, loss = knn_fit_predict_evaluate(X_train, X_test, y_train, y_test)
    trained_knns.append(knn)
    accuracies.append(acc)

accuracies = np.array(accuracies)
print('{}-fold cross-validation accuracy mean: {}'.format(splits, np.mean(accuracies)))
print('{}-fold cross-validation accuracy std: {}'.format(splits, np.std(accuracies, ddof=1)))
```

```
10-fold cross-validation accuracy mean: 0.9298429262812202
10-fold cross-validation accuracy std: 0.02691043470531831
```

ROC analysis

For a binary classification problem, where we have a positive (+) and a negative (−) class, we can obtain the confusion matrix of the expected and predicted results. The confusion matrix is organized as:

	Predicted +	**Predicted −**
Expected +	TP	FN
Expected −	FP	TN

From the above matrix, we can extract the following quantities:

- **True Positives (TP)**: the number of positive examples that were correctly predicted as positive;
- **True Negatives (TN)**: the number of negative examples that were correctly predicted as negative;
- **False Negatives (FN)**: the number of positive examples that were wrongly predicted as negative;
- **False Positives (FP)**: the number of negative examples that were wrongly predicted as positive.

Then, we can obtain two measures:

- **True Positive Rate (TPR)**: also known as sensibility. It measures the hit rate for the positive class. It is calculated as: $TPR = \frac{TP}{TP + FN}$;
- **False Positive Rate (FPR)**: also known as specificity. It measures the hit rate for the negative class. It is calculated as: $FPR = \frac{FP}{TN + FP}$.

Many classifiers output scores (or probabilities) when classifying an unseen example. These scores are usually thresholded in order to return a binary classification.

The ROC analysis consists of using several thresholds for the output scores of a classifier. Then, for each threshold, the respective TPR and FPR values can be calculated. By plotting the obtained (TPR, FPR) pairs, we obtain the ROC curve.

Finally, a commonly used measure to compare classifiers is the area under the ROC curve (ROC AUC). Such a measure lies between 0 and 1, with values close to 1 indicating better results.

We present an example below.

In [7]:

```
# Splitting X and y in train and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.34, stratify=y, random_state=seed)

# Creating and training a kNN classifier.
knn = KNeighborsClassifier(n_neighbors=DEFAULT_N_NEIGHBORS, weights='distance', metric='euclidean')
knn.fit(X_train, y_train)

# Predicting probability scores for the test set.
y_prob = knn.predict_proba(X_test)

# Calculatting False Positive Rate and True Positive Rate values for different thresholds.
# The first parameter consists of the expected labels.
# The second parameter consists of the predicted scores for the positive class. In this example
# the positive class is assumed to be the one with label = 1.
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob[:, 1])

# Calculating the area under the ROC curve.
roc_auc = auc(false_positive_rate, true_positive_rate)
```

Finally, we plot the ROC curve. The diagonal line of such a plot indicates a classifier with random predictions.

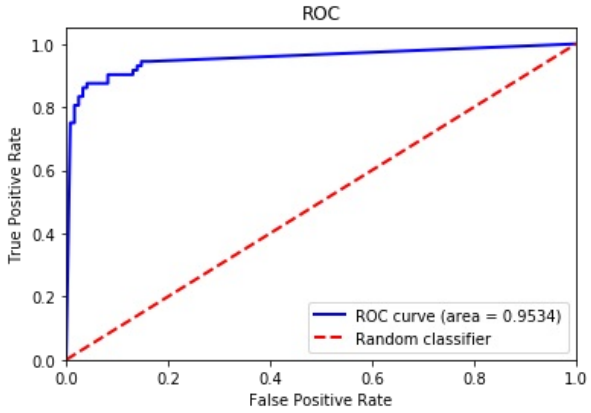
In [8]:

```
# setting linewidth = 2
lw = 2

plt.plot(false_positive_rate,
         true_positive_rate,
         color='blue',
         lw=lw,
         label='ROC curve (area = {:.4f})'.format(roc_auc))


plt.plot([0, 1], [0, 1], color='red', lw=lw, linestyle='--', label='Random classifier')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend(loc="lower right")

plt.show()
```



Hypothesis testing

Comparing two classifiers over multiple datasets

Usually, when comparing two classifiers (namely, f_1 and f_2), the null-hypothesis (H_0) states that their performances are equivalent. For this situation, Demšar (2006) recommends the Wilcoxon signed-rank test.

Next, we present an example extracted from (Demšar, 2006). In such an example, we have the area under the curve (AUC) for the C4.5 algorithm with the parameter m (the minimal number of examples in a leaf) equal to 0 and C4.5 with tunned m (C4.5+m) considering 14 datasets.

In [9]:

```
# Loading the example DataFrame.
performances = pd.read_csv('data/example_wilcoxon_demsar.csv')
performances
```

Out[9]:

	dataset	C4.5	C4.5+m
0	adult(sample)	0.763	0.768
1	breast_cancer	0.599	0.591
2	breast_cancer_wisconsin	0.954	0.971
3	cmc	0.628	0.661
4	ionosphere	0.882	0.888
5	iris	0.936	0.931
6	liver_disorders	0.661	0.668
7	lung_cancer	0.583	0.583
8	lymphography	0.775	0.838
9	mushroom	1.000	1.000
10	primary_tumor	0.940	0.962
11	rheum	0.619	0.666
12	voting	0.972	0.981
13	wine	0.957	0.978

In [10]:

```
# Getting C4.5 AUC values.
c45 = np.array(performances['C4.5'])

# Getting C4.5+m AUC values.
c45m = np.array(performances['C4.5+m'])

# Running Wilcoxon test. When zero_method='zsplit' the zero ranks are splitted between positive and negative ones.
wilcoxon(c45, c45m, zero_method='zsplit')
```

Out[10]:

```
WilcoxonResult(statistic=12.0, pvalue=0.010968496564224731)
```

The Wilcoxon signed-rank test outputs a p-value close to 0.01. If we consider a significance level (α) of 0.05 we can conclude that C4.5 and C4.5+m performances are not equivalent.

Comparing multiple classifiers over multiple datasets

The Wilcoxon signed-rank test was not designed to compare multiple random variables. So, when comparing multiple classifiers, an "intuitive" approach would be to apply the Wilcoxon test to all possible pairs. However, when multiple tests are conducted, some of them will reject the null hypothesis only by chance (Demšar, 2006).

For the comparison of multiple classifiers, Demšar (2006) recommends the Friedman test.

The Friedman test ranks the algorithms from best to worst on each dataset with respect to their performances. Its null-hypothesis (H_0) states that all algorithms are equivalent and their mean ranks are equal.

Next, we present an example extracted from (Demšar, 2006). In such an example, we have the AUC for four classifiers: C4.5 with $m = 0$ and the confidence interval parameter $cf = 0.25$, C4.5 with tunned m, C4.5 with tunned cf and C4.5 with both parameters tunned.

In [11]:

```
# Loading the example DataFrame.
performances = pd.read_csv('data/example_friedman_nemenyi_demsar.csv')
performances
```

Out[11]:

	dataset	C4.5	C4.5+m	C4.5+cf	C4.5+m+cf
0	adult(sample)	0.763	0.768	0.771	0.798
1	breast_cancer	0.599	0.591	0.590	0.569
2	breast_cancer_wisconsin	0.954	0.971	0.968	0.967
3	cmc	0.628	0.661	0.654	0.657
4	ionosphere	0.882	0.888	0.886	0.898
5	iris	0.936	0.931	0.916	0.931
6	liver_disorders	0.661	0.668	0.609	0.685
7	lung_cancer	0.583	0.583	0.563	0.625
8	lymphography	0.775	0.838	0.866	0.875
9	mushroom	1.000	1.000	1.000	1.000
10	primary_tumor	0.940	0.962	0.965	0.962
11	rheum	0.619	0.666	0.614	0.669
12	voting	0.972	0.981	0.975	0.975
13	wine	0.957	0.978	0.946	0.970

In [12]:

```
# First, we extract the algorithms names.
algorithms_names = performances.drop('dataset', axis=1).columns

# Then, we extract the performances as a numpy.ndarray.
performances_array = performances[algorithms_names].values

# Finally, we apply the Friedman test.
friedmanchisquare(*performances_array)
```

Out[12]:

```
FriedmanchisquareResult(statistic=51.285714285714278, pvalue=1.7912382226666844e-06)
```

The Friedman test outputs a very small p-value. For many significance levels (α) we can conclude that the performances of all algorithms are not equivalent.

Considering that the null-hypothesis was rejected, we usually have two scenarios for a post-hoc test (Demšar, 2006):

- All classifiers are compared to each other. In this case we apply the Nemenyi post-hoc test.
- All classifiers are compared to a control classifier. In this scenario we apply the Bonferroni-Dunn post-hoc test.

To perform both of the aformentioned post-hoc tests, we need the average rank of each algorithm,

In [13]:

```
# Calculating the ranks of the algorithms for each dataset. The value of p is multipled by -1
# because the rankdata method ranks from the smallest to the greatest performance values.
# Since we are considering AUC as our performance measure, we want larger values to be best ranked.
ranks = np.array([rankdata(-p) for p in performances_array])

# Calculating the average ranks.
average_ranks = np.mean(ranks, axis=0)

print('\n'.join('{} average rank: {}'.format(a, r) for a, r in zip(algorithms_names, average_ranks)))
```

```
C4.5 average rank: 3.142857142857143
C4.5+m average rank: 2.0
C4.5+cf average rank: 2.9285714285714284
C4.5+m+cf average rank: 1.9285714285714286
```

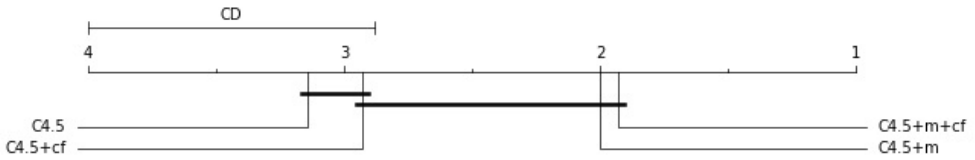
Then, we will calculate the critical differences and plot the results of each test (Nemenyi and Bonferroni-Dunn).

In [14]:

```
# This method computes the critical difference for Nemenyi test with alpha=0.1.
# For some reason, this method only accepts alpha='0.05' or alpha='0.1'.
cd = compute_CD(average_ranks,
                n=len(performances),
                alpha='0.1',
                test='nemenyi')

# This method generates the plot.
graph_ranks(average_ranks,
            names=algorithms_names,
            cd=cd,
            width=10,
            textspace=1.5,
            reverse=True)

plt.show()
```

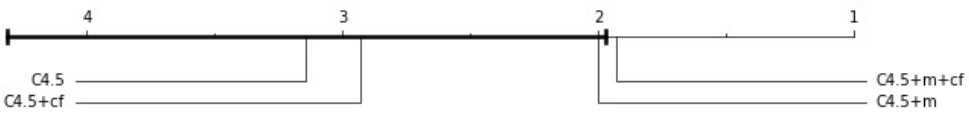


In [15]:

```
# This method computes the critical difference for Bonferroni-Dunn test with alpha=0.05.
# For some reason, this method only accepts alpha='0.05' or alpha='0.1'.
cd = compute_CD(average_ranks,
                n=len(performances),
                alpha='0.05',
                test='bonferroni-dunn')

# This method generates the plot.
graph_ranks(average_ranks,
            names=algorithms_names,
            cd=cd,
            cdmethod=0,
            width=10,
            textspace=1.5,
            reverse=True)

plt.show()
```



References

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7, 1-30.