In [1]:

```
from IPython.core.interactiveshell import InteractiveShell

# This file includes the functions to clean the times and shangai datasets
import data_cleaning as dc

InteractiveShell.ast_node_interactivity = "all"

%matplotlib inline

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
sns.set_context("notebook")
sns.set_context("poster")
```

Model Evaluation

We need to evaluate the performance of our machine learning models because of two main reason. First to be able to tune them deciding which interventions increase their performance and which ones don't. And secondly to have concrete knowlodge on how accurate they are and therefore to what extent can we trust them.

The first insight that we have to incorporate into our thinking of model evaluation is that by no means we can evaluate our model with the data that we used for training. Nowadays algorithms are very sophisticated and therefore prone to overfitting. It is therefore necessary to use new data, unseen by model, for its evaluation.

There are two main approaches. One, the most obvious, is to divide the data in a training and test set. We train the model with the train set and we use the test set to evaluate it. It is simple and works well if we have lots of data. However, if data is scarce, then we don't have enough diversity in the data and the evaluation could not be very accurate.

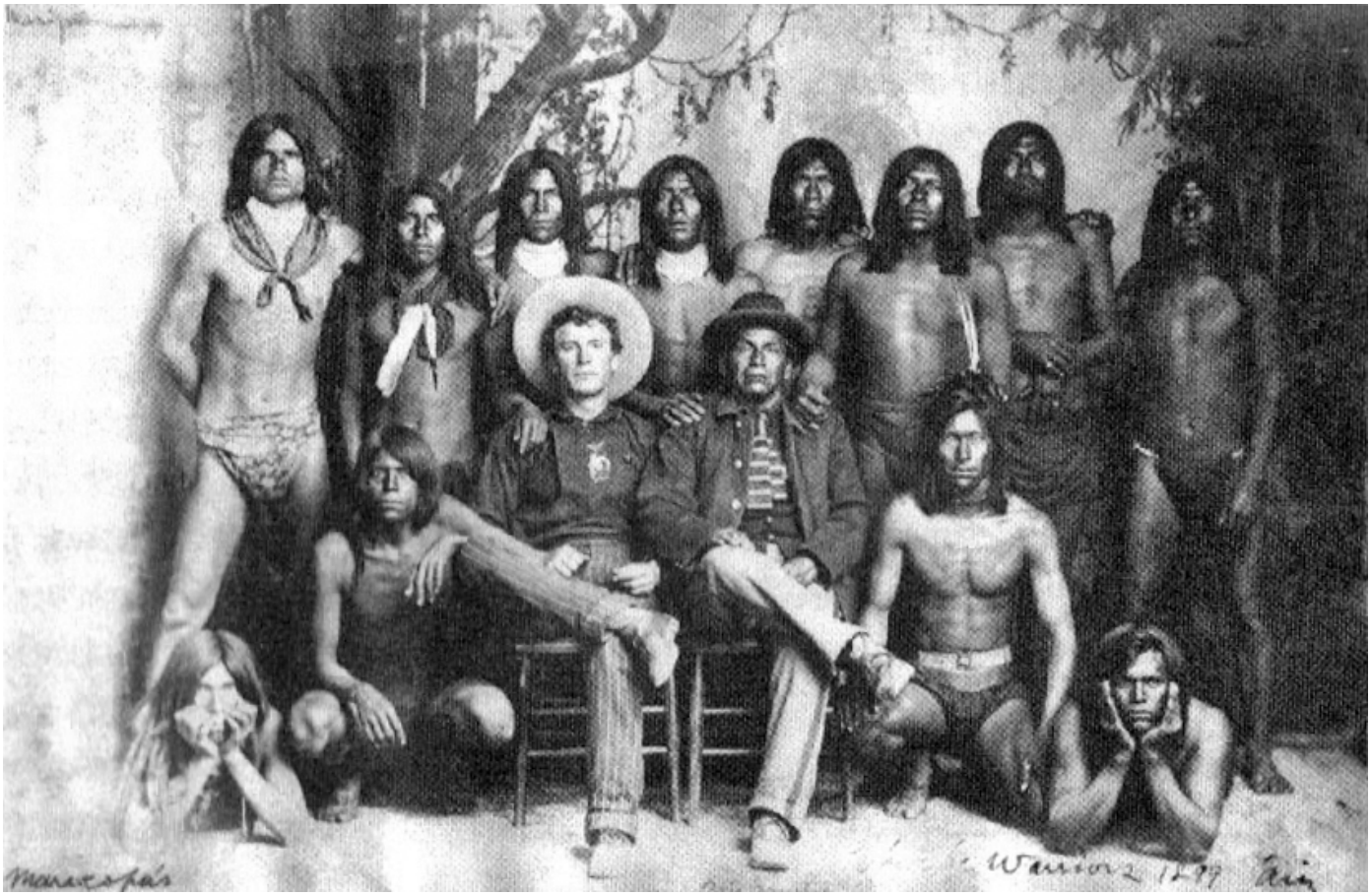
The second approach tries to solve this problem of evaluation, as accurate as we can, a model with a limited amount of data. As you can imagine, they consist on using sampling techniques with or without repetition in order to try to "augment" the amount of data available.

Once we have chosen the best hyperparameters and have the model ready for production, we train it with the whole data and put it in operational use.

We are going to look at four different techniques that we can use to split our data and create useful estimates of our models:

1) Train and test sets.
2) K-fold Cross-validation.
3) Leave one-out cross-validation.
4) Repeated random test-train splits.

Yes, we will use the Pima Indians onset of diabetes dataset.



In this exercise we will use one of the traditional Machine Learning dataset, the Pima Indians diabetes dataset.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Content The datasets consists of several medical predictor variables and one target variable, **Outcome**. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

- Pregnancies
- Glucose
- BloodPressure
- SkinThickness
- Insulin
- BMI
- DiabetesPedigreeFunction (scores de likelihood of diabetes based on family history)
- Age
- Outcome

In [2]:

```
# Load the Pima indians dataset and separate input and output components

from numpy import set_printoptions
set_printoptions(precision=3)

filename="pima-indians-diabetes.data.csv"
names=["pregnancies", "glucose", "pressure", "skin", "insulin", "bmi", "pedi", "age", "outc
p_indians=pd.read_csv(filename, names=names)
p_indians.head()

# First we separate into input and output components
array=p_indians.values
X=array[:,0:8]
Y=array[:,8]
X
pd.DataFrame(X).head()
```

Out[2]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Out[2]:

```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Out[2]:

	0	1	2	3	4	5	6	7
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0

Split into Train and Test Sets

A simple idea and also probably the most commonly used approach is to split our data into two sets. Use one for training and the other for testing. Normally a 70% of the data is used for training and 30% for testing, but of course these are arbitrary numbers and anything can be (e.g. 80% - 20% if the dataset is large).

The points in favor of this approach is that is simple and fast. It works well when datasets are large but also it is widely used as a first approximation. One important thing that must be taken into account is that the variance of both sets is similar, if not we can encounter unwanted surprises.

The downside is that we can have meaninful differences is the differences in variance are high and we that we take an important risk when the amount of data is small. Once the model is in production we may find that its performance has little in common with what we tested because the data that it encounters is really different.

The **train_test_split** module in scikit-learn is the one used for splitting the dataset.

In [3]:

```
# Split into Train and Test Sets
set_printoptions(precision=3)
p_indians.head()

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# we need to make it reproducible, so we use a seed for the pseudo-random
test_size=0.3
seed = 7

# the actual split
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=

# Let's do the log regresssion
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)

# Now let's find the accurary with the test split
result = model.score(X_test, Y_test)

print(f'Accuracy {result*100:5.3f}')
```

Out[3]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Out[3]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)

Accuracy 76.190
```

In [ ]:

In [ ]:

Mission 1

a) Change the distribution between Train and Test Sets. How does it affect accurarcy?

In [4]:

```
print("\033[1m" + "Test Set size at 10%" + "\033[0m")

# Test set size 10%

test_size=0.1
seed = 7

X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)
result = model.score(X_test, Y_test)
print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")
print()

print("\033[1m" + "Test Set size at 30%" + "\033[0m")

# Test set size 30%

test_size=0.3
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)
result = model.score(X_test, Y_test)
print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")
print()

print("\033[1m" + "Test Set size at 50%" + "\033[0m")

# Test set size 50%

test_size=0.5
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)
result = model.score(X_test, Y_test)
print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")
print()

print("\033[1m" + "Test Set size at 70%" + "\033[0m")

# Test set size 70%

test_size=0.7
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)
result = model.score(X_test, Y_test)
print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")
print()

print("\033[1m" + "Test Set size at 90%" + "\033[0m")

# Test set size 90%

test_size=0.9
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=test_size, random_state=
model = LogisticRegression(solver='liblinear')
model.fit(X_train,Y_train)
result = model.score(X_test, Y_test)
```

```
print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")

print()

print("The accuracy of the model decreases as we increase the test set size.")
print("The reason for this is that the train set is smaller, and hence the model has not be
print("Therefore, if the model is not as trained, than its accuracy will not be as good.")
```

Test Set size at 10%

Out[4]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Accuracy 83.117

Test Set size at 30%

Out[4]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Accuracy 76.190

Test Set size at 50%

Out[4]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Accuracy 77.344

Test Set size at 70%

Out[4]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

```
Accuracy 73.606

Test Set size at 90%
```

Out[4]:

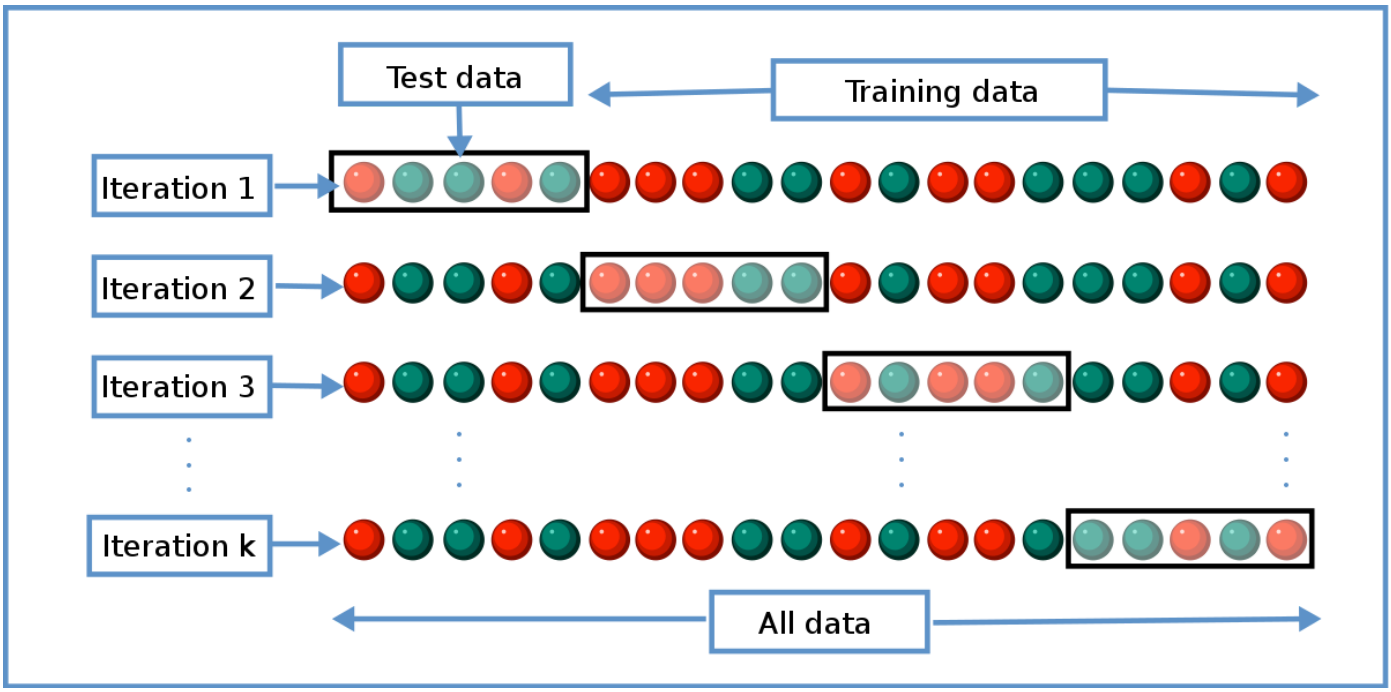
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

```
Accuracy 66.474

The accuracy of the model decreases as we increase the test set size.
The reason for this is that the train set is smaller, and hence the model ha
s not been as trained.
Therefore, if the model is not as trained, than its accuracy will not be as
 good.
```

In [ ]:

In [ ]:



K-fold Cross-Validation

The objective of k-fold cross-validation is to reduce the variance that we encounter when using the train-test split approach.

In this approach the available data is divided into k splits that are called folds (3, 5, 10 are common). We train and test the model k times. Each time we use k-1 folds for training and one fold for testing. Once we finish we use the mean of the evaluation measure together with its standard deviation as performance measure.

Obviously the dataset must be large enough to accommodate the process.

K-Fold Cross Validation uses the **KFold** class.

In [5]:

```
# K-fold Cross Validation

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

p_indians.head()

# KFold
splits=10
kfold=KFold(n_splits=splits, random_state=7, shuffle = True)

# Logistic regression
model = LogisticRegression(solver='liblinear')

# Obtain the performance measure - accuracy
results = cross_val_score(model, X, Y, cv=kfold)

print(f'Logistic regression, k-fold {splits:d} - Accuracy {results.mean()*100:5.3f}% ({resu
results
```

Out[5]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
Logistic regression, k-fold 10 - Accuracy 77.086% (5.091%)
```

Out[5]:

```
array([0.831, 0.714, 0.714, 0.779, 0.792, 0.766, 0.688, 0.857, 0.803,
       0.763])
```

In [ ]:

In [ ]:

Leave One Out Cross-Validation

What will happen if we take k-fold to the extreme? In this case we will have as many folds as points, so k will be equal to the number of points and the prediction will be done each time for the one point left.

This is an effort to make the most reasonable estimate possible given a dataset, it's called leave one out cross validation.

Obviously you pay a penalty in terms of computational expense and the standard deviation has more variance than with k-fold.

For Leave One Out Cross-Validation you use the **LeaveOneOut** class.

In [6]:

```
# Leave one out cross-validation

from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_val_score

from sklearn.linear_model import LogisticRegression

p_indians.head()

# Leave one out cross-validation
loo=LeaveOneOut()

# Logistic Regression
model = LogisticRegression(solver='liblinear')

#  performance
results = cross_val_score(model, X, Y, cv=loo)

print(f'Logistic regression, Leave one out - Accuracy {results.mean()*100:5.3f}% ({results.
```

Out[6]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
Logistic regression, Leave one out - Accuracy 76.823% (42.196%)
```

In [ ]:

In [ ]:

Repeated Random Test-Train Splits

Another approach is to apply repeatedly a train-test split. This way takes advantage of the train-test speed and the reduction of variance of cross validation at the same time.

A down side of the method is that we are including much of the same data, therefore results even if they look very nice, may not be realistic.

For Repeated Random Test-Train Splits you use the **ShuffleSplit** class.

In [7]:

```
# Repeated Random Test-Train Splits

from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score

from sklearn.linear_model import LogisticRegression

p_indians.head()

nrepeat=10
test_size=0.3
seed=7

shuffle=ShuffleSplit(n_splits=nrepeat, test_size=test_size, random_state=seed)

model = LogisticRegression(solver='liblinear')

res = cross_val_score(model, X, Y, cv=shuffle)

print(f'Log Regression - Repeated Test-Train {nrepeat:d} - Accuracy {res.mean()*100:5.3f}%
```

Out[7]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
Log Regression - Repeated Test-Train 10 - Accuracy 76.970% 1.366%
```

In [ ]:

In [ ]:

Which one to use and When?

First things first. K-fold is the gold-standard, if you are doing any serious work or presenting results to a digital educated audiende, please use k-fold and you'll avoid problems.

Train and Test is ok, and it is used for common quick & dirty work. As you have observed in the exercices if the dataset is moderately large, the differences are small. Certainly you avoid surprises with repeatedly using train-test or much better k-fold and your last model should be evaluated always this way, but train and test split is ok for model selection and hyperparameter tunning.

What about the rest? In all these techniques you try to balance accuracy in the estimated performance, evaluation speed and dataset size, they correspond to different bets in this balance.

You don't know what to do ... The staple is k-fold with 10-cross-validation, start there.

In [ ]:

In [ ]:

Mission 2

a) Using the Shangai Data and log regression for top-10, top-50 and top-100 evaluate the models with train-test split and k-fold-10.

b) Same for the data of the Times ranking.

In [8]:

```
# Train and test

def train_test(size, X, Y):

    print("\033[1m" + "Test Set size at " + str(size * 100) + "%" + "\033[0m")

    seed = 7

    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=size, random_state=s
    model = LogisticRegression(solver='liblinear')
    model.fit(X_train,Y_train)
    result = model.score(X_test, Y_test)
    print()
    print('\033[91m' + f'Accuracy {result*100:5.3f}' + "\033[0m")
    print()
    return model.fit(X_train,Y_train)


def kfold(n, X, Y):

    splits = n
    kfold = KFold(n_splits=splits, random_state=7, shuffle = True)

    # Logistic regression
    model = LogisticRegression(solver='liblinear')

    # Obtain the performance measure - accuracy
    results = cross_val_score(model, X, Y, cv=kfold)

    print(f'Logistic regression, k-fold {splits:d} - Accuracy {results.mean()*100:5.3f}% ({
    return results
```

In [9]:

```
# Function for cleaning shangai dataset

shan_10 = dc.shangai_clean(10).values
shan_50 = dc.shangai_clean(50).values
shan_100 = dc.shangai_clean(100).values

shan_x10 = shan_10[:,0:6]
shan_y10 = shan_10[:,6]

shan_x50 = shan_50[:,0:6]
shan_y50 = shan_50[:,6]

shan_x100 = shan_100[:,0:6]
shan_y100 = shan_100[:,6]

times_10 = dc.times_clean(10).values
times_50 = dc.times_clean(50).values
times_100 = dc.times_clean(100).values

times_x10 = times_10[:,0:9]
times_y10 = times_10[:,9]

times_x50 = times_50[:,0:9]
times_y50 = times_50[:,9]

times_x100 = times_100[:,0:9]
times_y100 = times_100[:,9]
```

In [10]:

```
print("\033[1m" + "Shangai training dataset" + "\033[0m")
print("")
print("\033[1m" + "Shangai top-10")
print("")

train_test(0.3, shan_x10, shan_y10)

print("")
print("\033[1m" + "Shangai top-50" + "\033[0m")
print("")

train_test(0.3, shan_x50, shan_y50)

print("")
print("\033[1m" + "Shangai top-100" + "\033[0m")
print("")

train_test(0.3, shan_x100, shan_y100)

print("")
print("\033[1m" + "Times training dataset" + "\033[0m")
print("")
print("\033[1m" + "Times top-10" + "\033[0m")
print("")

train_test(0.3, times_x10, times_y10)

print("")
print("\033[1m" + "Times top-50" + "\033[0m")
print("")

train_test(0.3, times_x50, times_y50)

print("")
print("\033[1m" + "Times top-100" + "\033[0m")
print("")

train_test(0.3, times_x100, times_y100)
```

Shangai training dataset

Shangai top-10

Test Set size at 30.0%

Accuracy 99.333

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Shangai top-50

Test Set size at 30.0%

Accuracy 96.667

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Shangai top-100

Test Set size at 30.0%

Accuracy 96.000

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Times training dataset

Times top-10

Test Set size at 30.0%

Accuracy 99.052

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Times top-50

Test Set size at 30.0%

Accuracy 96.209

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

Times top-100

Test Set size at 30.0%

Accuracy 97.156

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

In [11]:

```
print("\033[1m" + "Shangai kfold cross-validation" + "\033[0m")
print("")
print("")

print("\033[1m" + "Top 10" + "\033[0m")
print("")
kfold(10, shan_x10, shan_y10)
print("\033[1m" + "Top 50" + "\033[0m")
print("")
kfold(10, shan_x50, shan_y50)
print("\033[1m" + "Top 100" + "\033[0m")
print("")
kfold(10, shan_x100, shan_y100)


print("")
print("\033[1m" + "Times kfold cross-validation" + "\033[0m")
print("")
print("")

print("\033[1m" + "Top 10" + "\033[0m")
print("")
kfold(10, times_x10, times_y10)
print("\033[1m" + "Top 50" + "\033[0m")
print("")
kfold(10, times_x50, times_y50)
print("\033[1m" + "Top 100" + "\033[0m")
print("")
kfold(10, times_x100, times_y100)
```

Shangai kfold cross-validation

Top 10

Logistic regression, k-fold 10 - Accuracy 98.792% (1.332%)

Out[11]:

```
array([0.98, 1.  , 1.  , 1.  , 1.  , 0.96, 0.98, 1.  , 0.98, 0.98])
```

Top 50

Logistic regression, k-fold 10 - Accuracy 97.792% (1.076%)

Out[11]:

```
array([0.98, 0.98, 0.98, 1.  , 0.98, 0.98, 0.96, 0.96, 0.98, 0.98])
```

Top 100

Logistic regression, k-fold 10 - Accuracy 95.388% (2.685%)

Out[11]:

```
array([0.96 , 0.98 , 0.98 , 0.9  , 0.94 , 0.98 , 0.94 , 0.92 , 0.98 ,
       0.959])
```

Times kfold cross-validation

Top 10

```
Logistic regression, k-fold 10 - Accuracy 98.861% (1.534%)
```

Out[11]:

```
array([0.972, 1.   , 1.   , 0.986, 0.971, 1.   , 0.957, 1.   , 1.   ,
       1.   ])
```

Top 50

```
Logistic regression, k-fold 10 - Accuracy 97.439% (1.652%)
```

Out[11]:

```
array([0.986, 0.944, 0.986, 0.986, 0.971, 1.   , 0.957, 0.971, 0.957,
       0.986])
```

Top 100

```
Logistic regression, k-fold 10 - Accuracy 95.151% (2.576%)
```

Out[11]:

```
array([0.986, 0.958, 0.971, 0.957, 0.943, 0.943, 0.886, 0.943, 0.971,
       0.957])
```

In [ ]:

In [ ]: