showcase_sklearn

March 6, 2016

1 Showcase of scikit-learn

As the title says, this is only a showcase of scikit-learn as it was used in the thesis.
For actual documentation, refer to the scikit-learn documentation.

For classification with XGBoost, refer to showcase_xgboost.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import toolbox as tb
        import kaggleData as kD
        import matplotlib.patches as mpatches
```

In this showcase, we reproduce the best submissions we were able to achieve during the thesis by using the Python package scikit-learn.
These submissions are chosen by their performance on public AMS.
Following information is extracted from our recorded classification runs by using our tools, presented in showcase_toolbox.

```
In [2]: rec_data ,rec_header = tb.getRecord()
        rec_pubams=tb.sortByColumn(rec_data,3)

        gotIT = []
        best = []
        for row in reversed(rec_pubams):
            if row[0] not in gotIT:
                gotIT.append(row[0])
                best.append(row)
        best = np.array(best)
        best
```

```
Out[2]: array([['xgboost', 'header_all', '2.4321543', '3.66421262680941',
                '3.71268472156738', '997.632616996765', '57.4933519363403',
                'Threshold=0.145', 'steps_=2500', 'depth_=9', 'eta_=0.01',
                'subsample_=0.9', 'eval_1=auc', 'eval_2=ams@0.14', 'None', 'None',
                'None', 'None', 'None', 'None'],
               ['gbc', 'header_all', '0.869924', '3.28927069645223',
                '3.42808660853669', '8322.90483593940', '10.2775928974151',
                'threshold=0.6666', 'trees_=100', 'depth_=12', 'eta_=0.01',
                'subsample_=0.9', 'None', 'None', 'None', 'None', 'None', 'None',
                'None', 'None'],
               ['kNN', 'header_6', '0.81892', '3.18534579809683',
                '3.17144377327013', '1.25601601600646', '123.669127941131',
                'threshold=0.7777', 'k=297', 'p=1', 'None', 'None', 'None', 'None',
                'None', 'None', 'None', 'None', 'None', 'None'],
```

```
 ['log Reg', 'header_8', '0.73912', '2.04562322781762',
  '2.07029542090901', '3.67624711990356', '0.21499204635620',
  'threshold=0.4444', 'C=0.1', 'penalty=l2', 'None', 'None', 'None',
  'None', 'None', 'None', 'None', 'None', 'None', 'None'],
 ['log Reg CV', 'header_3', '0.73507', '1.96216755909995',
  '1.98149424825609', '33.3609240055084', '0.26706004142761',
  'threshold=0.4444', 'Cs=1', 'penalty=l1', "scoring = 'roc_a",
  'None', 'None', 'None', 'None', 'None', 'None', 'None', 'None',
  'None']],
dtype='<U16')
```

As we generate submissions in this showcase, we need to import the Kaggle dataset. We extract feature subsets if we need them to produce top submissions, for now we only generate the set with all features.

```
In [3]: csv_data,csv_header = kD.csvToArray()
        train_data,train_header,test_data,test_header=kD.getOriginalKaggleSets(csv_data,csv_header)
        sol_data,sol_header = kD.getSolutionKey(csv_data,csv_header)
        test_events = kD._extractFeature("EventId",test_data,csv_header).astype(float)
```

```
In [4]: train_all = train_data[:,1:-2].astype(float)
        train_labels = kD.translateLabels(train_data[:,-1],["Label"]).astype(float)
        train_weights = train_data[:,-2].astype(float)
        test_all = test_data[:,1:].astype(float)
        header_all = test_header[1:]
```

We want to test every classifier with toy data, so we need to generate it by using our tools.

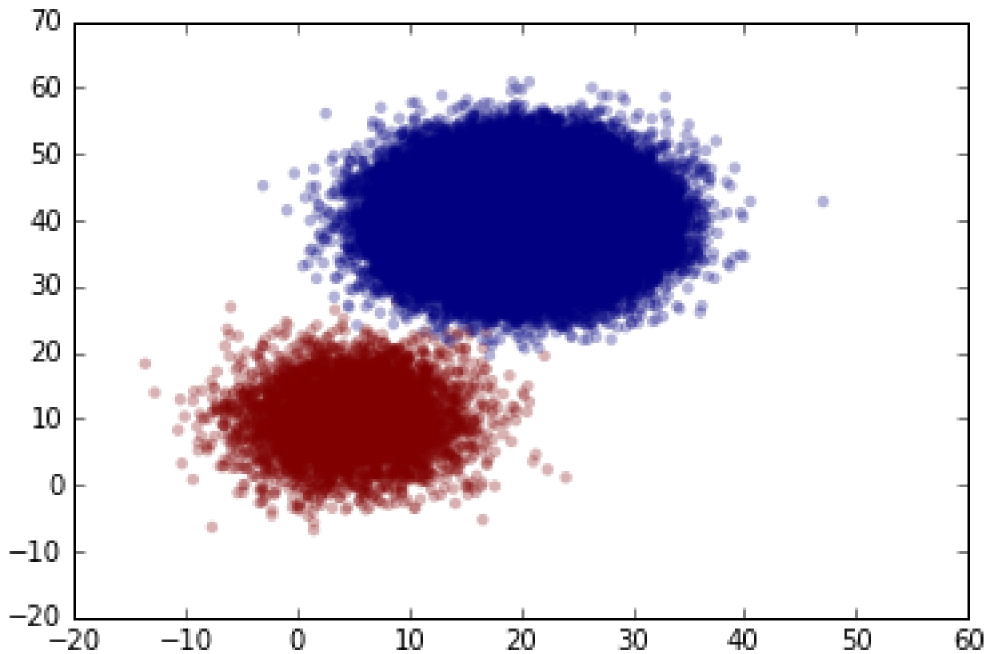
```
In [5]: n = 100000
        toy_data = tb.createToyData(n,dim = 4,s_prob = 0.05)
        toy_weights = toy_data[:,0]
        toy_labels = toy_data[:,1]
        x = toy_data[:,2:4]
```

To see the shape of the data, we visualize it.

```
In [6]: %pylab inline
        plt.scatter(x[:,0], x[:,1], edgecolor="", c=toy_labels, alpha=0.3)
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
Out[6]: <matplotlib.collections.PathCollection at 0x10177160>
```



For proper classification, we split our data into training and test sets and calculate the best AMS possible for this data.
The weights of our toy data are the random values we thresholded to separate our data into two classes. Therefore, all signal weights are less than or equal 0.05 and background is weighted more than 0.95.

```
In [7]: n_train = int(n/10)

        toy_train_x = x[:n_train]
        toy_test_x = x[n_train:]

        toy_train_labels = toy_labels[:n_train]
        toy_test_labels  = toy_labels[n_train:]

        toy_test_weights = toy_weights[n_train:]
```

```
In [8]: tb.calcMaxAMS(toy_test_weights,toy_test_labels);
```

```
Found 4518 signals.
Weightsums signal: 113.169951821 | background: 0
Maximum AMS possible with this Data: 19.80441049852333
```

1.1 Logistic Regression

```
In [9]: from sklearn import linear_model as linMod
```

Every classifier included in scikit-learn is used in three steps.
1. Initialize the classifier with wanted parameters. 2. Use the `fit(X,y)` function of this classifier, where `X` is the training data and `y` are its labels. 3. Use the `pred(X)` function of this classifier, where `X` is the test data.
For soft prediction, which has not been thresholded to separate classes yet, use `pred_proba(X)`.

In [10]:
```
logReg = linMod.LogisticRegression()
```

If we want to record the runtime, we can do this by using `time`.

In [11]:
```
import time
```

In [12]:
```
start = time.time()
logReg.fit(toy_train_x,toy_train_labels)
end = time.time()
print(end-start)
```

```
0.021059036254882812
```

In [13]:
```
pred = logReg.predict(toy_test_x)
```

We can calculate the accuracy the classifier achieves for a labeled data set.

In [14]:
```
logReg.score(toy_train_x,toy_train_labels)
```

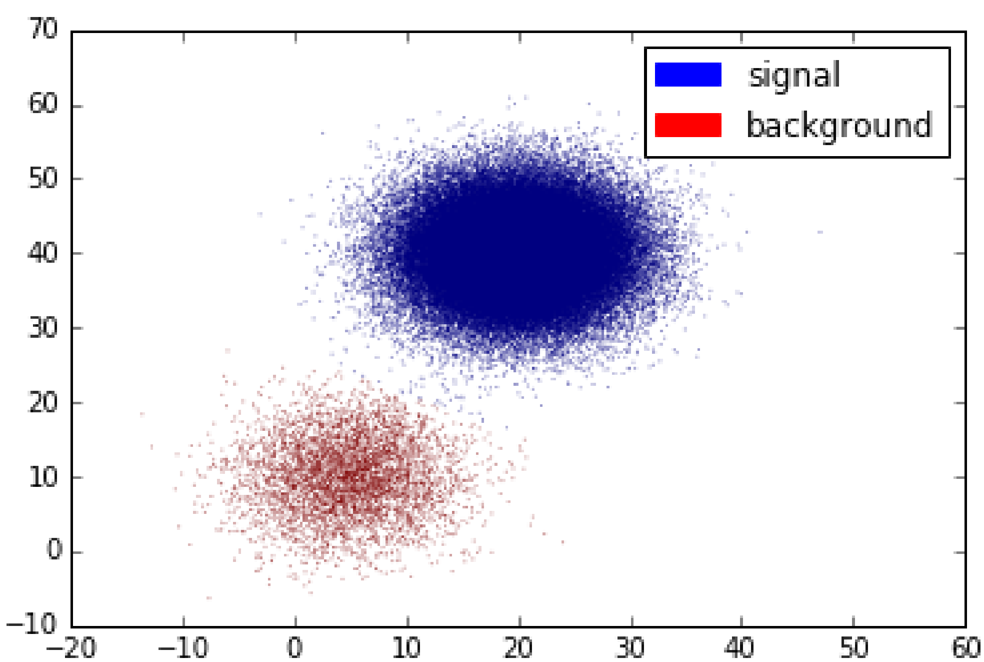
Out[14]:
```
0.99970000000000003
```

Also we are able to visualize the classified data.

In [15]:
```
%pylab inline

xData = toy_test_x[:,0]
yData = toy_test_x[:,1]

scat = plt.scatter(xData, yData, s=1, edgecolor="", c=pred, alpha=0.5)

blue_patch = mpatches.Patch(color='blue', label='signal')
red_patch = mpatches.Patch(color='red', label='background')
plt.legend(handles=[blue_patch,red_patch])
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[15]:
```
<matplotlib.legend.Legend at 0x155ab908>
```



Now we produce our submission that resulted to the best public AMS that was achieved by Logistic Regression.

```
In [16]: best[3]
```

```
Out[16]: array(['log Reg', 'header_8', '0.73912', '2.04562322781762',
                '2.07029542090901', '3.67624711990356', '0.21499204635620',
                'threshold=0.4444', 'C=0.1', 'penalty=l2', 'None', 'None', 'None',
                'None', 'None', 'None', 'None', 'None', 'None', 'None'],
               dtype='<U16')
```

We use feature set 8 (see Tab. 3 of the thesis.

```
In [17]: header_8,train_8,test_8 = kD.getFeatureSubset(train_data,test_data,train_header,test_header,8)
```

While testing a classifier, we perform cross-validation.

```
In [18]: from sklearn import cross_validation
```

Basically, we use `cross-validation.train_test_split()` to randomly split our training set and test the classifiers prediction.

```
In [19]: X_train, X_test, y_train, y_test = cross_validation.train_test_split(
             train_8, train_labels, test_size=0.4, random_state=0)
```

```
In [20]: logReg = linMod.LogisticRegression()
         logReg.fit(X_train,y_train)
         logReg.score(X_test,y_test)
```

```
Out[20]: 0.73926000000000003
```

To make recording of testing easier, we create methods. We use this strategy for every classifier we test. First, we write a method to perform logistic regression.

```
In [21]: def logisticReg(train_data,train_labels,test_data,C= 1.0,pen='l2',timed=False,n_jobs = 8,thres

             #setup
             logReg = linMod.LogisticRegression()
             logReg.set_params(C= C,
                               class_weight= None,
                               dual= False,
                               fit_intercept= True,
                               intercept_scaling= 1,
                               max_iter= 100,
                               multi_class= 'ovr',
                               n_jobs= n_jobs,
                               penalty= pen,
                               random_state= None,
                               solver= 'liblinear',
                               tol= 0.0001,
                               verbose= 0,
                               warm_start= False)

             #cross-validation
             X_train, X_test, y_train, y_test = cross_validation.train_test_split(
```

```
            train_data, train_labels, test_size=0.4, random_state=0)
        logReg.fit(X_train,y_train)
        cv_score = logReg.score(X_test, y_test)


        #performance testing
        time_train = 0.
        time_test = 0.
        if timed:
            start = time.time()

        #training classifier
        logReg.fit(train_data,train_labels)
        #we use sparsify for potential performance benefits, this makes the classification use spa
        logReg.sparsify()

        if timed:
            end = time.time()
            time_train = end - start
            start = time.time()

        #predict test_data, use threshold
        soft_pred = logReg.predict_proba(test_data)
        hard_pred = tb.customThreshold(soft_pred[:,1],thresh)

        if timed:
            end = time.time()
            time_test = end - start

        #
        del logReg
        return hard_pred,soft_pred,cv_score,time_train,time_test
```

Second, we want to achieve a best result by using automatic thresholding.
Then we save the run's performance and parameters.

```
In [22]: def runLogReg(train_data,train_labels,test_data,C=1.0,pen='l2',timed=True,featListName="not sp

             #make soft prediction
             soft_pred,cvs,time_train,time_test = logisticReg(train_data,train_labels,test_data,C=C,tim
             #threshold soft prediction
             hard_pred,maxAMS,maxThresh = tb.bestThreshold(soft_pred[:,1],sol_data)

             #calculate AMS and report it to user
             b_ams,v_ams = tb.calcSetAMS(hard_pred,sol_data)
             print("Public AMS:",b_ams[0],"|| Private AMS:",v_ams[0],"|| Threshold:", maxThresh)

             ##save run's stats
             res=np.empty((20,),dtype="<U16")
             res[:10]=["log Reg",
                     featListName,
                     str(cvs),
                     str(b_ams[0]),
                     str(v_ams[0]),
                     str(time_train),
```

```
                str(time_test),
                str("threshold="+str(maxThresh)),
                str("C="+str(C)),
                str("penalty="+pen)]
        res[10:]="None"
        tb.recordRun(res)

        return hard_pred,soft_pred[:,1]
```

```
In [23]: soft_pred = runLogReg(train_8,train_labels,test_8,C=0.1,pen='l2',timed=True,featListName="heade
```

```
Public AMS: 2.0456232278176207 || Private AMS: 2.0702954209090123 || Threshold: 0.444444444444
```

As we need to create a ranking for our submission, we use the soft prediction and the threshold chosen by `bestThreshold()`.
We call our submission subm_logReg.csv.

```
In [24]: tb.createSubmissionFile(soft_pred,fname="subm_logReg.csv",threshold=0.4444)
```

Scikit-learn provides a logistic regression class with built-in cross-validation. As its use is nearly identical to the standard class' and it failed to surpass it, we will not reproduce its best run.
For comparison, we present the recorded run in this place.

```
In [25]: best[4]
```

```
Out[25]: array(['log Reg CV', 'header_3', '0.73507', '1.96216755909995',
                '1.98149424825609', '33.3609240055084', '0.26706004142761',
                'threshold=0.4444', 'Cs=1', 'penalty=l1', "scoring = 'roc_a",
                'None', 'None', 'None', 'None', 'None', 'None', 'None', 'None',
                'None'],
               dtype='<U16')
```

1.2 K Nearest Neighbor

```
In [26]: from sklearn import neighbors
```

We run our first kNN classification on toy data.

```
In [27]: knn = neighbors.KNeighborsClassifier()
         knn.fit(toy_train_x,toy_train_labels)
         pred = knn.predict(toy_test_x)
         knn.score(toy_train_x,toy_train_labels)
```

```
Out[27]: 0.99970000000000003
```

The testing methods are:

```
In [28]: def kNN(train_data,train_labels,test_data,timed=False,n_jobs = 8,k = 20,p=2,thresh = 0.8):

             #setup
             neigh = neighbors.KNeighborsClassifier()
             neigh.set_params(algorithm = 'auto',
                              leaf_size = 30,
                              metric = 'minkowski',
                              metric_params = None,
                              n_jobs = n_jobs,
                              n_neighbors = k,
```

```
                     #p=1 <=> manhattan-distance
                     #p=2 <=> euclidian
                     p = p,
                     weights = 'distance')

    #cross-validation
    X_train, X_test, y_train, y_test = cross_validation.train_test_split(
        train_data, train_labels, test_size=0.4, random_state=0)
    neigh.fit(X_train,y_train)
    cv_score = neigh.score(X_test, y_test)

    #performance testing
    time_train = 0.
    time_test = 0.
    if timed:
        start = time.time()

    #model training
    neigh.fit(train_data,train_labels)

    if timed:
        end = time.time()
        time_train = end - start
        start = time.time()

    #predict test_data, use threshold
    soft_pred = neigh.predict_proba(test_data)
    hard_pred = tb.customThreshold(soft_pred[:,1],thresh)

    if timed:
        end = time.time()
        time_test = end - start

    #
    del neigh
    return hard_pred,soft_pred,cv_score,time_train,time_test
```

And:

```
In [29]: def run_kNN(train_data,train_labels,test_data,k=20,p=2,featListName="not specified"):

             #make soft prediction
             soft_pred,cvs,time_train,time_test = kNN(train_data,train_labels,test_data,timed=True,k=k,p

             #threshold soft prediction
             hard_pred,maxAMS,maxThresh = tb.bestThreshold(soft_pred[:,1],sol_data)

             #calculate AMS and report it to user
             b_ams,v_ams = tb.calcSetAMS(hard_pred,sol_data)
             print("Public AMS:",b_ams[0],"|| Private AMS:",v_ams[0],"|| Threshold:",maxThresh)

             ##save run's stats
             res=np.empty((20,),dtype="<U16")
             res[:10]=["kNN",
                     featListName,
```

```
                str(cvs),
                str(b_ams[0]),
                str(v_ams[0]),
                str(time_train),
                str(time_test),
                str("threshold="+str(maxThresh)),
                str("k="+str(k)),
                str("p="+str(p))
               ]
        res[10:]="None"
        tb.recordRun(res)

        return hard_pred,soft_pred[:,1]
```

Now we produce our submission that resulted to the best public AMS that was achieved by kNN.

```
In [30]: best[2]
```

```
Out[30]: array(['kNN', 'header_6', '0.81892', '3.18534579809683',
               '3.17144377327013', '1.25601601600646', '123.669127941131',
               'threshold=0.7777', 'k=297', 'p=1', 'None', 'None', 'None', 'None',
               'None', 'None', 'None', 'None', 'None', 'None'],
              dtype='<U16')
```

We use feature set 6 (see Tab. 3 of the thesis).

```
In [31]: header_6,train_6,test_6 = kD.getFeatureSubset(train_data,test_data,train_header,test_header,6)
```

```
In [32]: soft_pred = run_kNN(train_6,train_labels,test_6,k=297,p=1,featListName="header_6")[1]
```

```
Public AMS: 3.1853457980968303 || Private AMS: 3.171443773270133 || Threshold: 0.777777777778
```

We create the submission file subm_knn.csv .

```
In [33]: tb.createSubmissionFile(soft_pred,fname="subm_knn.csv",threshold=0.7777)
```

1.3 Gradient Boosting Classification

```
In [34]: import sklearn.ensemble
```

We produce a submission by using scikit-learn's implementation of gradient boosting.
We **do not** tune this classifier for optimal results, because we proceed to use XGBoost, as we follow the thesis.

```
In [35]: best[1]
```

```
Out[35]: array(['gbc', 'header_all', '0.869924', '3.28927069645223',
               '3.42808660853669', '8322.90483593940', '10.2775928974151',
               'threshold=0.6666', 'trees_=100', 'depth_=12', 'eta_=0.01',
               'subsample_=0.9', 'None', 'None', 'None', 'None', 'None', 'None',
               'None', 'None'],
              dtype='<U16')
```

First, we make a test run on toy data and achieve a perfect score:

```
In [36]: gbc = sklearn.ensemble.GradientBoostingClassifier()
         gbc.fit(toy_train_x,toy_train_labels)
         pred = gbc.predict(toy_test_x)
         gbc.score(toy_train_x,toy_train_labels)
```

Out[36]: 1.0

For creating the submission file, we proceed just like for previous classification. We create a more general testing and a recording method.

```
In [37]: def gradientBoosting(train_data,label,test_data,timed=False,trees=100,depth=9,eta=0.1,threshol
             #setup
             gbc = sklearn.ensemble.GradientBoostingClassifier()
             gbc.set_params(init=None,
                        learning_rate=eta,
                        loss='deviance',
                        max_depth=depth,
                        max_features=None,
                        max_leaf_nodes=None,
                        min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        n_estimators=trees,
                        presort='auto',
                        random_state=None,
                        subsample=0.9,
                        verbose=0,
                        warm_start=False)

             #performance testing
             time_train = 0.
             time_test = 0.
             if timed:
                 start = time.time()

             #training
             gbc.fit(train_data,label)

             if timed:
                 end = time.time()
                 time_train = end - start

             #We did not perform actual cross-validation for gbc we substitue the cv score with gbc's a
             cvs = gbc.score(train_data,label)

             if timed:
                 start = time.time()

             #predict test_data
             soft_pred = gbc.predict_proba(test_data)

             if timed:
                 end = time.time()
                 time_test = end - start

             return soft_pred,cvs,time_train,time_test

In [38]: def run_gbc(train_data,label,test_data,timed=False,trees=100,depth=9,eta=0.1,featListName="not

             #make soft prediction
```

```
        soft_pred,cvs,time_train,time_test = gradientBoosting(train_data,train_labels,test_data,ti

        #threshold soft prediction
        hard_pred,maxAMS,maxThresh = tb.bestThreshold(soft_pred[:,1],sol_data)

        #calculate AMS and report it to user
        b_ams,v_ams = tb.calcSetAMS(hard_pred,sol_data)
        print("Public AMS:",b_ams[0],"|| Private AMS:",v_ams[0],"|| Threshold:", maxThresh)

        #save run's stats
        res=np.empty((20,),dtype="<U16")
        res[:12]=["gbc",
                featListName,
                str(cvs),
                str(b_ams[0]),
                str(v_ams[0]),
                str(time_train),
                str(time_test),
                str("threshold="+str(maxThresh)),
                str("trees_="+str(trees)),
                str("depth_="+str(depth)),
                str("eta_="+str(eta)),
                str("subsample_=0.9")
                ]

        res[12:]="None"
        tb.recordRun(res)

        return hard_pred,soft_pred[:,1]
```

If you intend to run the following methods on your computer, keep in mind that this run took **over an hour** to terminate.

```
In [ ]: #soft_pref = run_gbc(train_all,train_labels,test_all,timed=True,trees=100,depth=12,eta=0.01,fea
```

We create the submission file subm_gbc.csv

```
In [ ]: #tb.createSubmissionFile(soft_pred,fname="subm_gbc.csv",threshold=0.6666)
```