homework04

February 14, 2021

0.1 REI602M Machine Learning - Homework 4

0.1.1 Due: *Sunday* 14.2.2021

Objectives: Parameter tuning, Ensemble tree classifiers, Feature importance, Stacking, Preprocessing, performance metrics, scikit-learn and pandas.

Name: Alexander Guðmundsson, **email:** alg35@hi.is, **collaborators:** (if any)

Please provide your solutions by filling in the appropriate cells in this notebook, creating new cells as needed. Hand in your solution on Gradescope, taking care to locate the appropriate page numbers in the PDF document. Make sure that you are familiar with the course rules on collaboration (encouraged) and copying (very, very, bad).

0.1.2 1. [Pre-processing and parameter tuning in an SVM classifier, 20 points]

The Statlog data set is an old benchmark in machine learning. It contains data from satellite images and the aim is to predict land type (red soil, cotton crop etc). There are 36 integer valued features and 6 classes. The file `sat.trn` contains the 4435 training examples and `sat.tst` contains 2000 test examples. Your task is to obtain an RBF-SVM classifier which obtains high classification accuracy on this data set.

a) Evaluate the accuracy of a RBF-SVM obtained with default values for C and γ.

b) Scale the training set prior to training an RBF-SVM (scale the test data accordingly) and repeat the task from a).

c) Use cross-validation on the (scaled) training set to identify optimal values of C and γ. You should start with a coarse grid and then do another run with a finer grid. Logarithmically spaced grid values are often used. Evaluate the accuracy of the resulting classifier by retraining using the best parameter values and report the error on the test set (why is the cross-validation error not a good estimate of the true classifier error here?)

d) Using randomized search for hyper-parameter values has been shown to be more efficient than traditional grid search. Instead of evaluating parameter values at regular intervals, the values are sampled from a distribution over possible parameter values. This enables considerable time savings (exhaustive grid search is expensive), or the identification of better parameter values for a given computational budget. Repeat the parameter search using `RandomizedSearchCV`. You can either fix the budget (n_iter parameter) so that the cost is comparable to the exhaustive grid search in c), or set the budget to e.g. 10% of the cost in iii). Report the results of the best classifier.

Summarize briefly your findings from a) to d).

Comments:

1) Description of the data set: https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite)
2) For scaling the data, see: `preprocessing.StandardScaler` in scikit.
3) The parameters C and γ in SVMs are called *hyper-parameters* since they are considered to be fixed during optimization of the model parameters (θ-values). The procedure of selecting hyper-parameter values is referred to as *model selection* and is typically performed by training the model for several different values of hyper-parameters, and evaluating the resulting model on a (cross-)validation set and finally selecting the values that give the best results.
4) An exhaustive search of parameter combinations on a grid is called is referred to as a *grid-search*. The `GridSearchCV` class in scikit-learn makes this easy to do. For details see the scikit documentation, sections 3.1 (Cross-validation: evaluating estimator performance), 3.2 (Tuning the hyper-parameters of an estimator) and the "Parameter estimation using grid search with cross-validation" example.
5) The paper *Random Search for Hyper-Parameter Optimization* by Bergstra and Bengio describes why randomized search of hyperparamter values is an efficient strategy. https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

[191]:
```
import numpy as np
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler as std_scl
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV



#init train
trainData = np.genfromtxt(fname = "sat.trn")
X_train = trainData[:,:-1]
y_train = trainData[:,-1]

n,p = X_train.shape

X_train = np.c_[np.ones(n),X_train]

#init test
testData = np.genfromtxt(fname = "sat.tst")
X_test = testData[:,:-1]
y_test = testData[:,-1]

n_test,p_test = X_test.shape

X_test = np.c_[np.ones(n_test),X_test]


#Evaluate the accuracy of a RBF-SVM obtained with default values
```

```
ev = SVC().fit(X_train,y_train)
print("train set accuracy with default values of C and  = ", ev.score(X_train,␣
 ↪y_train))
print("test set accuracy with default values of C and  = ",ev.score(X_test,␣
 ↪y_test))


#scale the data
X_train_scale = std_scl().fit(X_train).transform(X_train)
X_test_scale = std_scl().fit(X_test).transform(X_test)



#Evaluate the accuracy of a RBF-SVM with scaled data
ev = SVC().fit(X_train_scale,y_train)
print("scaled train set accuracy with default values of C and  = ", ev.
 ↪score(X_train_scale, y_train))
print("scaled test set accuracy with default values of C and  = ", ev.
 ↪score(X_test_scale, y_test))


#cross validating to find C and
grid = {
    'C': [1, 10, 100, 1000],
    'gamma': [0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
              0.7, 0.8, 0.9, 0.95, 0.99]
}

grid_search_cv = GridSearchCV(estimator = ev, param_grid = grid, cv = 3)
grid_search_cv = grid_search_cv.fit(X_train_scale, y_train)

best_params = grid_search_cv.best_params_
print("the best parameters for C and  = ",best_params)

#Classifying with the best C and
ev = SVC(C = best_params['C'], gamma = best_params['gamma']).fit(X_train_scale,␣
 ↪y_train)
print("scaled train set accuracy with best values of C and  = ", ev.
 ↪score(X_train_scale, y_train))
print("scaled test set accuracy with best values of C and  = ", ev.
 ↪score(X_test_scale, y_test))
```

```
train set accuracy with default values of C and  =  0.8962795941375423
test set accuracy with default values of C and  =  0.88
scaled train set accuracy with default values of C and  =  0.9095828635851184
scaled test set accuracy with default values of C and  =  0.8895
```

```
the best parameters for C and   =  {'C': 1, 'gamma': 0.1}
scaled train set accuracy with best values of C and   =  0.9341600901916572
scaled test set accuracy with best values of C and   =  0.91
```

[199]:
```
from sklearn.model_selection import RandomizedSearchCV
RS = RandomizedSearchCV(ev,grid)
RS.fit(X_train_scale, y_train)
```

[199]:
```
RandomizedSearchCV(estimator=SVC(C=1, gamma=0.1),
                   param_distributions={'C': [1, 10, 100, 1000],
                                        'gamma': [0.01, 0.05, 0.1, 0.2, 0.3,
                                                  0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
                                                  0.95, 0.99]})
```

When looking at the results in a-c, we can see that the accuracy increses, cross validation error is not the same as true classifier error, we are cross validating for the train set not the test set.

0.1.3 2. [Random Forests and feature selection, 20 points]

Quantitative Structure - Activity Relationship (QSAR) models relate the activity of chemical compounds to their structural properties. The activity can represent e.g. the potency of a drug or its toxicity. The structural properties may contain basic information such as i) the fraction of carbon atoms in the compound, ii) the spatial arrangement of atoms in the compound and iii) quantities computed from quantum mechanical simulations (*ab-initio* calculations).

The QSAR biodegradation Data Set `biodeg.csv` contains 41 molecular descriptors for two groups of compounds, those that are readily biodegradable (RB) and those that are not (NRB). The data set has 356 examples in the RB class and 699 in the NRB class, i.e. it is somewhat unbalanced.

a) Using `sklearn.ensemble.RandomForestClassifier`, obtain a classifier for predicting whether a given compound is readily biodegradable or not. Use a random 80/20 train/test set split for evaluting the performance of your classifier. Report the confusion matrix for the test set and calculate the accuracy of the classifier together with *sensitivity* and *specificity* (see below).

b) List the names of the 10 *most useful* features for the classification task (see below). Retrain a Random forests classifier using only the 10 most useful features and report sensitivity, specificity and accuracy. How does the performance compare to the classifier trained on the full feature set in a)?

Comments:

1) The file `biodeg_desc.txt` containes the feature names. A description of the data set can be found here: https://archive.ics.uci.edu/ml/datasets/QSAR+biodegradation

2) Use `sklearn.model_selection.train_test_split` to create a train/test split.

3) A correctly predicted RB example is said to be a *true positive* and a correctly predicted NRB examples is said to be a *true negative*. When an NRB example is incorrectly predicted as RB it is a *false positive*. False negatives are defined analogously.

The *sensitivity* of a binary classifier is defined as TP/(TP+FN) and the *specificity* is defined as TN/(TN+FP) where TP is the number of true positives etc. These values are conveniently obtained

from a confusion matrix, e.g. via `sklearn.metrics.confusion_matrix`

4) The feature importance measure provided by the Random Forests implementation in scikit has significant drawbacks. Therefore you will be using a so-called permutation importance measure (see problem 1 for a description). This is implemented in `sklearn.inspection.permutation_importance`.

5) Sidenote: Repeatedly retraining a classifier with smaller and smaller number of top features until the validation error (or out-of-bag error) starts to increase can easily lead to overfitting.

[93]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import pandas as pd


#CreateMatrix(y_actual, y_predicted)
#loops through the y values and compares them
#returns two (nxn) confusion matrixes with count and ratio
def createMatrix(y, y_pred):
    total = len(y)
    TP = 0
    TN = 0
    FP = 0
    FN = 0

    for i in range(len(y)):
        if y[i] == 1 and y_pred[i] == 1:
            TP += 1
        if y[i] == 0 and y_pred[i] == 0:
            TN += 1
        if y[i] == 0 and y_pred[i] == 1:
            FP += 1
        if y[i] == 1 and y_pred[i] == 0:
            FN += 1

    matrix = np.array([[TP,FP],
                       [FN,TN]])


    matrixR = np.array([[float(TP/total),float(FP/total)],
                        [float(FN/total),TN/total]])

    return matrix,matrixR

#matrixCalc(matrix)
#calculates sensitivity, specifity, accuracy and precision of an matrix
#returns dict with keys as "sensitivity", "specificity", "accuracy" and␣
 ↪"precision"
```

```
#with the calculations as values
def matrixCalc(matrix):
    TP = matrix[0,0]
    FP = matrix[0,1]
    FN = matrix[1,0]
    TN = matrix[1,1]

    calc = {}
    calc["sensitivity"] = float(TP/(TP+FN))
    calc["specificity"] = float(TN/(TN + FP))
    calc["accuracy"] = float((TP+TN)/(TP + TN + FP + FN))
    calc["precision"] = float(TP/(TP + FP))

    return calc

#Insert data
Data = np.genfromtxt(fname = "biodeg.csv", delimiter = ';')
y = Data[:,-1]
X = Data[:,0:-1]
n,r = X.shape
X = np.c_[np.ones(n),X]

#classifying
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
random_forest_classifier = RandomForestClassifier().fit(X_train, y_train)
y_pred = random_forest_classifier.predict(X_test)

#finding accuracy

confMatrix, confMatrixR = createMatrix(y_test, y_pred)

#drawing the matrix
data = {'actual value 1':confMatrix[:,0],
       'actual value 0':confMatrix[:,1]}

dataR = {'actual value 1':confMatrixR[:,0],
       'actual value 0':confMatrixR[:,1]}

table = pd.DataFrame(data, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix")
print(table,'\n\n')
tableR = pd.DataFrame(dataR, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix ration")
print(tableR,'\n\n')
```

```
#defining value for calculations
calc = matrixCalc(confMatrix)
prevAcc = ([x + "=" + str(calc[x]) for x in calc])
print(prevAcc)
```

```
confusion matrix
                   actual value 1  actual value 0
predicted value 1              61              10
predicted value 0              14             126


confusion matrix ration
                   actual value 1  actual value 0
predicted value 1        0.289100        0.047393
predicted value 0        0.066351        0.597156


['sensitivity=0.8133333333333334', 'specificity=0.9264705882352942',
'accuracy=0.8862559241706162', 'precision=0.8591549295774648']
```

[94]:
```
from sklearn.inspection import permutation_importance

headers = []

#getting the names of each label
text = open("biodeg_desc.txt", "r")
for i in range(len(X[0])):
    line = text.readline()
    if line == "":
        break
    split1 = line.split(')',1)
    split2 = split1[1].split(':')
    headers.append(split2[0])

top = permutation_importance(random_forest_classifier,X, y)

importances_mean = top["importances_std"]
importance_sorted = np.argsort(importances_mean)[::-1][:10] #sort and find top␣
 ↪10 indexes

print("top ten most useful features are: ", [headers[x] for x in␣
 ↪importance_sorted])

X_new = []

#creating new X with only the top features
for i in importance_sorted:
```

```
    X_new.append(X[:,i])

X_new = np.array(X_new)
X_new = np.transpose(X_new)

#classifying
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_size=0.2,␣
 ↪random_state=100)
random_forest_classifier = RandomForestClassifier().fit(X_train, y_train)
y_pred = random_forest_classifier.predict(X_test)

#finding accuracy

confMatrix, confMatrixR = createMatrix(y_test, y_pred)


#drawing the matrix
data = {'actual value 1':confMatrix[:,0],
       'actual value 0':confMatrix[:,1]}

dataR = {'actual value 1':confMatrixR[:,0],
       'actual value 0':confMatrixR[:,1]}

table = pd.DataFrame(data, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix")
print(table,'\n\n')
tableR = pd.DataFrame(dataR, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix ration")
print(tableR,'\n\n')


#defining value for calculations
calc = matrixCalc(confMatrix)
print([x + "=" + str(calc[x]) for x in calc])
print('\n\n')

print("previous =", prevAcc)
```

```
top ten most useful features are:  [' Psi_i_A', ' J_Dz(e)', ' nHDon', '
HyWi_B(m)', ' Psi_i_1d', ' F01[N-N]', ' F03[C-O]', ' F03[C-N]', ' nHM', ' nN-N']
confusion matrix
                  actual value 1  actual value 0
predicted value 1              56               6
predicted value 0              25             124


confusion matrix ration
                  actual value 1  actual value 0
```

```
predicted value 1        0.265403        0.028436
predicted value 0        0.118483        0.587678
```

```
['sensitivity=0.691358024691358', 'specificity=0.9538461538461539',
'accuracy=0.8530805687203792', 'precision=0.9032258064516129']
```

```
previous = ['sensitivity=0.8133333333333334', 'specificity=0.9264705882352942',
'accuracy=0.8862559241706162', 'precision=0.8591549295774648']
```

there we can see that the sensitivity and accuracy went down but precision and specificity went up

0.1.4 3. [Stacked regression models, 30 points]

In this problem you will construct a *stacked* two-stage regression model for a subset of the Million Song Database (MSD). The data set contains audio features for approximately 500K songs. Each song is represented by 90 features describing its "timbre" that are derived from the sampled recordings. The task is to predict the release year of a song.

A two-stage stacking model has several regression models in stage 1, all trained on the same data set. Predictions from stage 1 models form a new (derived) data set which is used as input to a single regression model in stage 2. This model "blends" predictions from the stage 1 models to create a final prediction, hopefully more accurate than the individual stage 1 predictions.

Your stacked regression model will employ Lasso, ExtraTrees, Random Forests and Gradient boosted trees in stage 1 and a linear regression model in stage 2. Training and testing are performed as follows:

Training: Train each model on the training set, using default parameters to begin with, but increase the number of trees for Extra Trees and Random Forests. Construct a training data set for the stage 2 model by sending the *validation* set (not the original training set) through each of the stage 1 models, resulting in an `n_val` by 4 matrix X_2 of prediction values. Train a linear regression model for stage 2 on (X_2, y_{val}).

Testing: Send the test data though all the models in stage 1 to obtain an `n_test` by 4 matrix. The stage 2 linear regression model is used to predict the data in this matrix to obtain the final predictions.

Start by creating a histogram of the number of songs per year in the data set to obtain insight into how realistic this prediction task is.

a) [20 points] Report the mean-squared error of the individual stage 1 models on the test set and the corresponding R^2 coefficient. Construct the stacked regression model described above, report its mean-squared error and R^2 coefficient on the test set.

b) [10 points] Answer the following questions:

c) Are the individual models doing a good job on the prediction task? (Consider the root-mean squared error).

ii) Are the individual classifiers failing in some obvious way (failure modes)?

iii) Is the stacking procedure worth the extra effort in your opinion? Why or why not?

iv) Why is it not a good idea to use the original training set to construct the X_2 data set for the stage 2 regression model?

Comments:

1) Download the subset of the Million Song Databse from here (210 MB): http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD# (mirror: https://notendur.hi.is/steinng/kennsla/2021/ml/data/YearPredictionMSD.zip)

2) Use the train, validation and test partitions of the data defined in `load_msd.py`

`import load_msd as lmsd X_train, y_train, X_val, y_val, X_test, y_test = lmsd.get_data(ntrain=10000)`

3) For Extra Trees and Random Forests you can set `n_jobs=-1` to use multiple cores/processors for training and prediction.

[133]:
```
import load_msd as lmsd
X_train, y_train, X_val, y_val, X_test, y_test = lmsd.get_data(ntrain=10000)
```

[189]:
```
from sklearn.ensemble import StackingRegressor
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import Lasso
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LinearRegression
lvl1_reg = [
    ('etc', ExtraTreesClassifier(n_jobs=-1)),
    ('rfc', RandomForestClassifier(n_jobs=-1)),
    ('gbc', GradientBoostingClassifier()),
]
lvl1_class = [
    ('lm', Lasso())
]


lvl2 = LinearRegression()




#model = StackingClassifier(estimators = lvl1_reg)
#model.fit(X_train, y_train)
#print(model.score(X_test, y_test))
```

```
    ␣
↪---------------------------------------------------------------------------

      KeyboardInterrupt                         Traceback (most recent call␣
↪last)

      <ipython-input-189-a5664467fd34> in <module>
       23
       24 model = StackingClassifier(estimators = lvl1_reg)
  ---> 25 model.fit(X_train, y_train)
       26 print(model.score(X_test, y_test))
       27


      ~\anaconda3\lib\site-packages\sklearn\ensemble\_stacking.py in fit(self,␣
↪X, y, sample_weight)
      423         self._le = LabelEncoder().fit(y)
      424         self.classes_ = self._le.classes_
  --> 425         return super().fit(X, self._le.transform(y), sample_weight)
      426
      427     @if_delegate_has_method(delegate='final_estimator_')


      ~\anaconda3\lib\site-packages\sklearn\ensemble\_stacking.py in fit(self,␣
↪X, y, sample_weight)
      143         # base estimators will be used in transform, predict, and
      144         # predict_proba. They are exposed publicly.
  --> 145         self.estimators_ = Parallel(n_jobs=self.n_jobs)(
      146             delayed(_fit_single_estimator)(clone(est), X, y,␣
↪sample_weight)
      147             for est in all_estimators if est != 'drop'


      ~\anaconda3\lib\site-packages\joblib\parallel.py in __call__(self,␣
↪iterable)
     1030                 self._iterating = self._original_iterator is not None
     1031
  -> 1032             while self.dispatch_one_batch(iterator):
     1033                 pass
     1034


      ~\anaconda3\lib\site-packages\joblib\parallel.py in␣
↪dispatch_one_batch(self, iterator)
      845                 return False
      846             else:
  --> 847                 self._dispatch(tasks)
```

```
        848                 return True
        849
```

```
        ~\anaconda3\lib\site-packages\joblib\parallel.py in _dispatch(self,␣
↪batch)
        763         with self._lock:
        764             job_idx = len(self._jobs)
    --> 765             job = self._backend.apply_async(batch, callback=cb)
        766             # A job can complete so quickly than its callback is
        767             # called before we get here, causing self._jobs to
```

```
        ~\anaconda3\lib\site-packages\joblib\_parallel_backends.py in␣
↪apply_async(self, func, callback)
        206     def apply_async(self, func, callback=None):
        207         """Schedule a func to be run"""
    --> 208         result = ImmediateResult(func)
        209         if callback:
        210             callback(result)
```

```
        ~\anaconda3\lib\site-packages\joblib\_parallel_backends.py in␣
↪__init__(self, batch)
        570         # Don't delay the application, to avoid keeping the input
        571         # arguments in memory
    --> 572         self.results = batch()
        573
        574     def get(self):
```

```
        ~\anaconda3\lib\site-packages\joblib\parallel.py in __call__(self)
        250         # change the default number of processes to -1
        251         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    --> 252             return [func(*args, **kwargs)
        253                     for func, args, kwargs in self.items]
        254
```

```
        ~\anaconda3\lib\site-packages\joblib\parallel.py in <listcomp>(.0)
        250         # change the default number of processes to -1
        251         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    --> 252             return [func(*args, **kwargs)
        253                     for func, args, kwargs in self.items]
        254
```

```
      ~\anaconda3\lib\site-packages\sklearn\ensemble\_base.py in␣
↪_fit_single_estimator(estimator, X, y, sample_weight, message_clsname, message)
       38     else:
       39         with _print_elapsed_time(message_clsname, message):
  ---> 40             estimator.fit(X, y)
       41     return estimator
       42


      ~\anaconda3\lib\site-packages\sklearn\ensemble\_gb.py in fit(self, X, y,␣
↪sample_weight, monitor)
      496
      497         # fit the boosting stages
  --> 498         n_stages = self._fit_stages(
      499             X, y, raw_predictions, sample_weight, self._rng, X_val,␣
↪y_val,
      500             sample_weight_val, begin_at_stage, monitor, X_idx_sorted)


      ~\anaconda3\lib\site-packages\sklearn\ensemble\_gb.py in␣
↪_fit_stages(self, X, y, raw_predictions, sample_weight, random_state, X_val,␣
↪y_val, sample_weight_val, begin_at_stage, monitor, X_idx_sorted)
      553
      554             # fit next stage of trees
  --> 555             raw_predictions = self._fit_stage(
      556                 i, X, y, raw_predictions, sample_weight, sample_mask,
      557                 random_state, X_idx_sorted, X_csc, X_csr)


      ~\anaconda3\lib\site-packages\sklearn\ensemble\_gb.py in␣
↪_fit_stage(self, i, X, y, raw_predictions, sample_weight, sample_mask,␣
↪random_state, X_idx_sorted, X_csc, X_csr)
      209
      210             X = X_csr if X_csr is not None else X
  --> 211             tree.fit(X, residual, sample_weight=sample_weight,
      212                      check_input=False, X_idx_sorted=X_idx_sorted)
      213


      ~\anaconda3\lib\site-packages\sklearn\tree\_classes.py in fit(self, X,␣
↪y, sample_weight, check_input, X_idx_sorted)
     1240         """
     1241
  -> 1242         super().fit(
     1243             X, y,
     1244             sample_weight=sample_weight,
```

```
       ~\anaconda3\lib\site-packages\sklearn\tree\_classes.py in fit(self, X,␣
↪y, sample_weight, check_input, X_idx_sorted)
       373                                            min_impurity_split)
       374
   --> 375         builder.build(self.tree_, X, y, sample_weight, X_idx_sorted)
       376
       377         if self.n_outputs_ == 1 and is_classifier(self):


       KeyboardInterrupt:
```

0.1.5 4. [Preprocessing, performance metrics, 30 points]

In this problem you will construct a predictive model for Telemarketing. The data comes from a telemarketing campaign in Portugal where the goal was to get clients to subscribe to long-term savings deposits. The predictive model is to be used to identify customers that are likely to subscribe, based on personal information, economic indicators, whether the client has been contacted before etc. This should make the campaign more effective and reduce marketing costs.

The data is in file `bank-additional-full.csv` with a brief description in `bank-additional-names.txt`. The data contains a mixture of continuous and categorical features, with categorical data in text format. Some preprocessing is therefore needed before you can use it with scikit-learn algorithms.

The data is time ordered which means that randomly splitting it into training and test sets amounts to peeking into the future and will provide too optimistic estimates of model performance. This is therefore not a suitable evaluation strategy. In situations like this, one uses historical data to train a model and predicts data from the current period. To simulate this scenario you will use the most recent (last) 4000 samples for testing. You then use the remaining samples for training (or a subset thereof).

Train a Random Forests or an Extra Trees classifier and evaluate it on the test set using an appropriate performance metric (see below). The selection of metric should take into account whether the classes are balanced or not, as well as the goal of the prediction task. Do you think your model would be useful in practice? Why or why not?

Comments:

1) This is an open ended problem. There is no single correct answer.

2) The data set is described in some detail here: https://archive.ics.uci.edu/ml/datasets/bank+marketing and a previous attempt at predictive modeling in: https://www.sciencedirect.com/science/article/pii/S01679

3) To convert the data into a matrix format suitable for scikit-learn, it is probably easiest to use the Python Data Analysis Library (pandas) package. You load the data using

```
python import pandas as pd bank_df=pd.read_csv('bank-additional-full.csv',sep=';')
```

You can iterate over the features using e.g.

```
python for col in bank_df.columns:     if bank_df[col].dtype == object:
print("Categorical: ",col)     else:         print("Numerical: ", col)
```

The output variable (y) is 1 if a customer subscribes and 0 otherwise.

Start by using only the numerical data. Then add the categorical data gradually. More data does not always help.

The simplest conversion of categories to integers is called label encoding. In pandas:

```
python bank_df['y']=bank_df['y'].astype('category') bank_df['y']=bank_df['y'].cat.codes
y=bank_df['y'].values bank_df=bank_df.drop('y',axis=1) # Remove output variable
```

or using scikit-learn instead:

```
python from sklearn.preprocessing import LabelBinarizer lb = LabelBinarizer() y =
lb.fit_transform(bank_df['y'])[:,0]
```

Label encoding of a feature assumes that the feature values have a natural ordering (are ordinal). This has some drawbacks as detailed in the lecture notes and one-hot-encoding is generally preferred. This is most conveniently done by using `pd.get_dummies` with `drop_first=True`. For this particular data set, direct application of one-hot-encoding does not necessarily improve performance.

4) In addition to `sklearn.metrics.confusion_matrix` which can be used to derive sensitivity, specificity and accuracy, the `sklearn.metrics.classification_report` class provides performance metrics called precision recall and F-score (see Wikipedia for details).

[121]:
```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier



#importing the data
bank_df = pd.read_csv('bank-additional-full.csv',sep=';')


#changing categorical labels to integers
for col in bank_df.columns:
    if bank_df[col].dtype == object:
        bank_df[col]=bank_df[col].astype('category')
        bank_df[col]=bank_df[col].cat.codes

#prepering data
y = bank_df['y'].values
bank_df = bank_df.drop('y', axis=1)
X_train = bank_df[0:-4000]
y_train = y[0:-4000]
X_test = bank_df[-4000:]
y_test = y[-4000:]
```

```
#classifying the data

random_forest_classifier = RandomForestClassifier().fit(X_train, y_train)
y_pred = random_forest_classifier.predict(X_test)

#finding accuracy

confMatrix, confMatrixR = createMatrix(y_test, y_pred)

#drawing the matrix
data = {'actual value 1':confMatrix[:,0],
       'actual value 0':confMatrix[:,1]}

dataR = {'actual value 1':confMatrixR[:,0],
       'actual value 0':confMatrixR[:,1]}

table = pd.DataFrame(data, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix")
print(table,'\n\n')
tableR = pd.DataFrame(dataR, index=['predicted value 1', 'predicted value 0'])
print("confusion matrix ration")
print(tableR,'\n\n')


#defining value for calculations
calc = matrixCalc(confMatrix)
prevAcc = ([x + "=" + str(calc[x]) for x in calc])
print(prevAcc)
```

```
confusion matrix
                  actual value 1  actual value 0
predicted value 1             432             137
predicted value 0            1423            2008


confusion matrix ration
                  actual value 1  actual value 0
predicted value 1         0.10800         0.03425
predicted value 0         0.35575         0.50200


['sensitivity=0.23288409703504043', 'specificity=0.9361305361305361',
'accuracy=0.61', 'precision=0.7592267135325131']
```

As can be seen above, accuracy is pretty bad, let's take a look at the data to see why that might be

[130]:
```
from sklearn.metrics import classification_report as c_rep

report = c_rep(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.59      0.94      0.72      2145
           1       0.76      0.23      0.36      1855

    accuracy                           0.61      4000
   macro avg       0.67      0.58      0.54      4000
weighted avg       0.67      0.61      0.55      4000
```

We can see that the recall and f1 score are very low for classification = 1, let's take a look at the original data to see why it is.

[131]:
```
print("total numbers of y instances as 1 =",sum(y==1))
print("total numbers of y instances as 0 =",sum(y==0))
```

```
total numbers of y instances as 1 = 4640
total numbers of y instances as 0 = 36548
```

we can see that the output data is very unbalanced and when classifying you first need to balance the data.

[ ]:
```

```