ECE 805 - Machine Learning

Tutorial 4: Online Learning (13 February 2023)

Contents

1. Data pre-processing

1.1 Standardization, or mean removal and variance scaling

> Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.

> In practice we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

```
from sklearn import preprocessing
import numpy as np

X_train = np.array([[ 1., -1.,  2.],[ 2.,  0.,  0.],[ 0.,  1., -1.]])
scaler = preprocessing.StandardScaler().fit(X_train)

X_scaled = scaler.transform(X_train)
X_scaled
```

```
array([[ 0.        , -1.22474487,  1.33630621],
       [ 1.22474487,  0.        , -0.26726124],
       [-1.22474487,  1.22474487, -1.06904497]])
```

1.1.1 Scaling features to a range

> An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively.

> The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

```
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])

min_max_scaler = preprocessing.MinMaxScaler()
X_train_minmax = min_max_scaler.fit_transform(X_train)
X_train_minmax
```

```
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:

```
X_test = np.array([[-3., -1.,  4.]])
X_test_minmax = min_max_scaler.transform(X_test)
X_test_minmax
```

```
array([[-1.5       ,  0.        ,  1.66666667]])
```

1.2 Normalization

> Normalization is the process of scaling individual samples to have unit norm. This process can be useful if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pair of samples. This assumption is the base of the Vector Space Model often used in text classification and clustering contexts. The function normalize provides a quick and easy way to perform this operation on a single array-like dataset, either using the l1, l2, or max norms:

```
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
X_normalized = preprocessing.normalize(X, norm='l2')

X_normalized
```

```
array([[ 0.40824829, -0.40824829,  0.81649658],
       [ 1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.70710678, -0.70710678]])
```

> The preprocessing module further provides a utility class Normalizer that implements the same operation using the Transformer API (even though the fit method is useless in this case: the class is stateless as this operation treats samples independently).

```
normalizer = preprocessing.Normalizer().fit(X)  # fit does nothing
normalizer.transform(X)
normalizer.transform([[-1.,  1., 0.]])
```

```
array([[-0.70710678,  0.70710678,  0.        ]])
```

1.3 Encoding

> Often features are not given as continuous values but categorical. For example a person could have features ["male", "female"], ["from Europe", "from US", "from Asia"], ["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"]. Such features can be efficiently coded as integers, for instance ["male", "from US", "uses Internet Explorer"] could be expressed as [0, 1, 3] while ["female", "from Asia", "uses Chrome"] would be [1, 2, 1].

> To convert categorical features to such integer codes, we can use the OrdinalEncoder. This estimator transforms each categorical feature to one new feature of integers (0 to n_categories - 1):

```
enc = preprocessing.OrdinalEncoder()
X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
enc.fit(X)

enc.transform([['female', 'from US', 'uses Safari']])
```

```
array([[0., 1., 1.]])
```

1.4 Discretization

> The SimpleImputer class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located. This class also allows for different missing values encodings.

> The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean value of the columns (axis 0) that contain the missing values:

1.4.1 K-bins discretization

> KBinsDiscretizer discretizes features into k bins.By default the output is one-hot encoded into a sparse matrix (See Encoding categorical features) and this can be configured with the encode parameter. For each feature, the bin edges are computed during fit and together with the number of bins, they will define the intervals. Therefore, for the current example, these intervals are defined as:

feature 1:

feature 2:

feature 3:

Based on these bin intervals, X is transformed as follows:

```
X = np.array([[ -3., 5., 15 ],
              [  0., 6., 14 ],
              [  6., 3., 11 ]])
est = preprocessing.KBinsDiscretizer(n_bins=[3, 2, 2], encode='ordinal').fit(X)
est.transform(X)
```

```
array([[0., 1., 1.],
       [1., 1., 1.],
       [2., 0., 0.]])
```

1.4.2 Feature binarization

> Feature binarization is the process of thresholding numerical features to get boolean values. This can be useful for downstream probabilistic estimators that make assumption that the input data is distributed according to a multi-variate Bernoulli distribution. For instance, this is the case for the BernoulliRBM.

> It is also common among the text processing community to use binary feature values (probably to simplify the probabilistic reasoning) even if normalized counts (a.k.a. term frequencies) or TF-IDF valued features often perform slightly better in practice.

> As for the Normalizer, the utility class Binarizer is meant to be used in the early stages of Pipeline. The fit method does nothing as each sample is treated independently of others:

```
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]

binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing

binarizer.transform(X)
```

```
array([[1., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.]])
```

1.5 Imputation of missing values

> One type of imputation algorithm is univariate, which imputes values in the i-th feature dimension using only non-missing values in that feature dimension (e.g. impute.SimpleImputer). By contrast, multivariate imputation algorithms use the entire set of available feature dimensions to estimate the missing values (e.g. impute.IterativeImputer).

1.5.1 Univariate feature imputation

> The SimpleImputer class provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located. This class also allows for different missing values encodings.

> The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean value of the columns (axis 0) that contain the missing values:

```
import numpy as np
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit([[1, 2], [np.nan, 3], [7, 6]])
X = [[np.nan, 2], [6, np.nan], [7, 6]]
print(imp.transform(X))
```

```
[[4.         2.        ]
 [6.         3.66666667]
 [7.         6.        ]]
```

1.5.2 Multivariate feature imputation

> A more sophisticated approach is to use the IterativeImputer class, which models each feature with missing values as a function of other features, and uses that estimate for imputation. It does so in an iterated round-robin fashion: at each step, a feature column is designated as output y and the other feature columns are treated as inputs X. A regressor is fit on (X, y) for known y. Then, the regressor is used to predict the missing values of y. This is done for each feature in an iterative fashion, and then is repeated for max_iter imputation rounds. The results of the final imputation round are returned.

```
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
imp = IterativeImputer(max_iter=10, random_state=0)
imp.fit([[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]])

X_test = [[np.nan, 2], [6, np.nan], [np.nan, 6]]
# the model learns that the second feature is double the first
print(np.round(imp.transform(X_test)))
```

```
[[ 1.  2.]
 [ 6. 12.]
 [ 3.  6.]]
```

2. Import libraries

The following code allows us to import functions from other Jupyter Notebooks.

```
import io, os, sys, types
import nbformat
from IPython import get_ipython
from IPython.core.interactiveshell import InteractiveShell


def find_notebook(fullname, path=None):
    """find a notebook, given its fully qualified name and an optional path

    This turns "foo.bar" into "foo/bar.ipynb"
    and tries turning "Foo_Bar" into "Foo Bar" if Foo_Bar
    does not exist.
    """
    name = fullname.rsplit('.', 1)[-1]
    if not path:
        path = ['']
    for d in path:
        nb_path = os.path.join(d, name + ".ipynb")
        if os.path.isfile(nb_path):
            return nb_path
        # let import Notebook_Name find "Notebook Name.ipynb"
        nb_path = nb_path.replace("_", " ")
        if os.path.isfile(nb_path):
            return nb_path


class NotebookLoader(object):
    """Module Loader for IPython Notebooks"""
    def __init__(self, path=None):
        self.shell = InteractiveShell.instance()
        self.path = path

    def load_module(self, fullname):
        """import a notebook as a module"""
        path = find_notebook(fullname, self.path)

        print ("importing notebook from %s" % path)

        # load the notebook object
        nb = nbformat.read(path, as_version=4)


        # create the module and add it to sys.modules
        # if name in sys.modules:
        #    return sys.modules[name]
        mod = types.ModuleType(fullname)
        mod.__file__ = path
        mod.__loader__ = self
        mod.__dict__['get_ipython'] = get_ipython
        sys.modules[fullname] = mod

        # extra work to ensure that magics that would affect the user_ns
        # actually affect the notebook module's ns
        save_user_ns = self.shell.user_ns
        self.shell.user_ns = mod.__dict__

        try:
          for cell in nb.cells:
            if cell.cell_type == 'code':
                # transform the input to executable Python
                code = self.shell.input_transformer_manager.transform_cell(cell.source)
```

```
                # run the code in themodule
                exec(code, mod.__dict__)
        finally:
            self.shell.user_ns = save_user_ns
        return mod


class NotebookFinder(object):
    """Module finder that locates IPython Notebooks"""
    def __init__(self):
        self.loaders = {}

    def find_module(self, fullname, path=None):
        nb_path = find_notebook(fullname, path)
        if not nb_path:
            return

        key = path
        if path:
            # lists aren't hashable
            key = os.path.sep.join(path)

        if key not in self.loaders:
            self.loaders[key] = NotebookLoader(path)
        return self.loaders[key]


sys.meta_path.append(NotebookFinder())
```

Import the libraries we will be using.

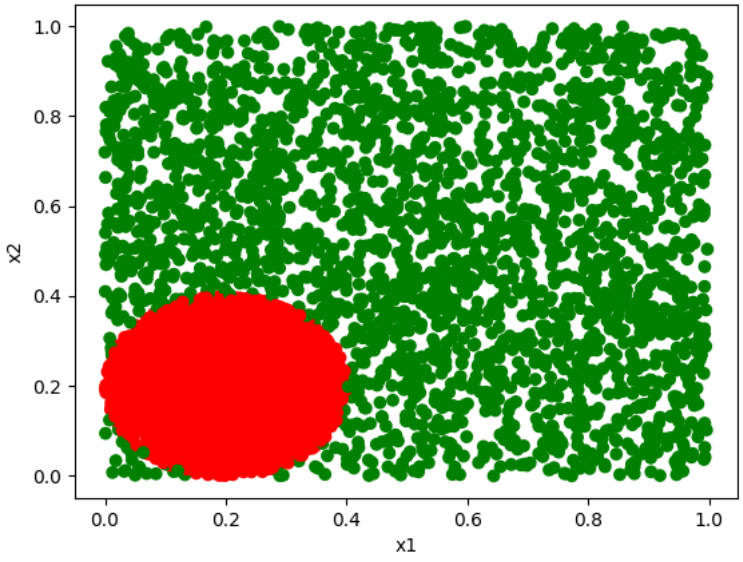
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from classifiers import Classifier
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
```

3. Datasets

3.1 Synthetic datasets

We will use the **Circle** synthetic dataset.

Circle dataset




```
def load_data(dataset_name):
    data = None

    if 'circle' in dataset name:
```

```
        data_circle = pd.read_csv('circle.csv').values

        if 'drift' not in dataset_name:
            data = data_circle
        else:
            data_circle_drifted = pd.read_csv('circle_drifted.csv').values
            data = np.r_[data_circle[:2500, :], data_circle_drifted[2500:, :]]

    elif 'cancer' in dataset_name:
        data = load_breast_cancer()
        data_x = data.data
        data_y = data.target

        # rescale data_x
        sc = StandardScaler()
        data_x = sc.fit_transform(data_x)
        data = np.c_[data_x, data_y]

    return data
```

Let's see some example points from the "Circle" dataset.

```
load_data(dataset_name='circle')[:5,:]
```

```
    array([[0.0043977 , 0.19470524, 1.        ],
           [0.29614308, 0.23157425, 1.        ],
           [0.81230884, 0.85540437, 0.        ],
           [0.54622862, 0.77018062, 0.        ],
           [0.2499697 , 0.24141536, 1.        ]])
```

3.2 Real-world dataset

Wisconsin Breast Cancer

The classification task is to predict whether a breast mass is malicious or benign. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. FNA is a diagnostic procedure used to investigate masses where a needle is inserted into the mass for sampling of cells. There exist 569 images with 30 features that describe characteristics of the cell nuclei present in the image (e.g. radius, perimeter, area). More details can be found here.

```
load_data(dataset_name='cancer')[:2,:]
```

```
    array([[ 1.09706398e+00, -2.07333501e+00,  1.26993369e+00,
             9.84374905e-01,  1.56846633e+00,  3.28351467e+00,
             2.65287398e+00,  2.53247522e+00,  2.21751501e+00,
             2.25574689e+00,  2.48973393e+00, -5.65265059e-01,
             2.83303087e+00,  2.48757756e+00, -2.14001647e-01,
             1.31686157e+00,  7.24026158e-01,  6.60819941e-01,
             1.14875667e+00,  9.07083081e-01,  1.88668963e+00,
            -1.35929347e+00,  2.30360062e+00,  2.00123749e+00,
             1.30768627e+00,  2.61666502e+00,  2.10952635e+00,
             2.29607613e+00,  2.75062224e+00,  1.93701461e+00,
             0.00000000e+00],
           [ 1.82982061e+00, -3.53632408e-01,  1.68595471e+00,
             1.90870825e+00, -8.26962447e-01, -4.87071673e-01,
            -2.38458552e-02,  5.48144156e-01,  1.39236330e-03,
            -8.68652457e-01,  4.99254601e-01, -8.76243603e-01,
             2.63326966e-01,  7.42401948e-01, -6.05350847e-01,
            -6.92926270e-01, -4.40780058e-01,  2.60162067e-01,
            -8.05450380e-01, -9.94437403e-02,  1.80592744e+00,
            -3.69203222e-01,  1.53512599e+00,  1.89048899e+00,
            -3.75611957e-01, -4.30444219e-01, -1.46748968e-01,
             1.08708430e+00, -2.43889668e-01,  2.81189987e-01,
             0.00000000e+00]])
```

4. Online Gradient Descent

4.1 Online learning framework

In online learning there is no separation between a training phase and a testing phase. Instead, learning is performed in a sequence of consecutive rounds. Specifically, each time we receive an example, it is first considered as a test example which we attempt to predict its class. The true class label is then received which can help us to improve our prediction for future examples.

Algorithm 1 Online learning

1: h^0: classifier ▷ e.g. Logistic Regression, Neural Network
2: **for** each time step $t \in [1, \infty)$ **do**
3: receive example x^t
4: predict class label $\hat{y}^t = h^{t-1}(x^t)$
5: receive class label y^t
6: calculate loss $L = l(y^t, \hat{y}^t)$
7: update classifier $h^t = h^{t-1}.train(L)$

Algorithm 1 Online Gradient Descent for Neural Network

1: h^0: initialise neural network
2: $\lambda = 0.99$ ▷ fading factor for prequential accuracy
3: $preq\_acc\_s^0 = 0.0$
4: $preq\_acc\_n^0 = 0.0$
5: **for** each time step $t \in [1, \infty)$ **do**
6: receive example x^t
7: predict class label $\hat{y}^t = h^{t-1}(x^t)$ using forward computation
8: receive class label y^t
9: calculate logistic loss $L = l(y^t, \hat{y}^t)$
10: update classifier $h^t = h^{t-1}.train(L)$ using backward computation
11: update $preq\_acc\_s^t = correct + \lambda * preq\_acc\_s^{t-1}$
12: update $preq\_acc\_n^t = 1.0 + \lambda * preq\_acc\_n^{t-1}$
13: calculate $preq\_acc = \frac{preq\_acc\_s}{preq\_acc\_n}$ ▷ for visualisation

Remarks

Online supervised learning (Line 5): It is assumed that the ground truth will be provided after the prediction and before the arrival of the next instance. This might be fine in some applications, but not in all. Alternative paradigms are *online active* and *online unsupervised* learning.

One-pass learning (Line 6): It occurs when the classifier considers only *the most recent* example. Some online algorithms also consider previously observed examples e.g. using a sliding window ("abrupt forgetting"). Some others use all previous examples but give a low priority to old ones ("gradual forgetting").

Incremental learning (Line 7): It occurs when the same classifier is constantly updated. In some cases, a classifier can be discarded and a new one starts learning from scratch.

4.2 Online Gradient Descent

Online Gradient Descent is almost identical to **Stochastic** Gradient Descent. Below is the pseudocode for **Online Logistic Regression**.

Algorithm 2 Online Gradient Descent for Logistic Regression

1: h_θ^0: logistic regression
2: **for** each time step $t \in [1, \infty)$ **do**
3: receive example x^t
4: predict class label $\hat{y}^t = sigm((\theta^{t-1})^T x^t)$
5: receive class label y^t
6: calculate logistic loss $L = l(y^t, \hat{y}^t) = -y^t log(\hat{y}^t) - (1 - y^t) log(1 - \hat{y}^t)$
7: update classifier $\left\{ \theta_j^t = \theta_j^{t-1} - \alpha \frac{\partial L}{\partial \theta_j^{t-1}} \right\}$

Auxiliary functions

```
def update_preq_metric(s_prev, n_prev, correct, fading_factor=0.99):
    s = correct + fading_factor * s_prev
    n = 1.0 + fading_factor * n_prev
    metric = s / n

    return s, n, metric


def plot_results(simulation_time, preq_general_accs, dataset_name, flag_store=False):
    plt.figure(figsize=(5, 5))
    plt.xlabel('Time Step', fontsize=10, weight='bold')
    plt.xticks(np.arange(0, simulation_time + 50, int(simulation_time / 5)), fontsize=15)
    plt.xlim(0, simulation_time)
    plt.ylabel('Accuracy', fontsize=10, weight='bold')
    plt.yticks(np.arange(0.0, 1.1, 0.2), fontsize=15)
    plt.ylim(0, 1)
    plt.grid(linestyle='dotted')

    plt.plot(range(simulation_time), preq_general_accs, linewidth=1.0)
    if flag_store:
        plt.savefig('results/' + dataset_name + '.png')
    plt.show()
```

Algorithm

```
def ogd(data_params, cls_params, t_drift=2500, flag_store_results=False):
    # get data parameters
    dataset_name = data_params['dataset_name']
    num_features = data_params['num_features']

    #  load data
    data = load_data(dataset_name)

    # create classifier
    cls = Classifier(num_features, cls_params)

    # prequential evaluation
    preq_general_accs = []
    preq_general_acc_n = 0.0
    preq_general_acc_s = 0.0

    # start
    simulation_time = data.shape[0]
    for t in range(simulation_time):
        if t % 500 == 0:
            print('Time step: ', t)

        if t == t_drift and 'drift' in dataset_name:
            preq_general_acc_n = 0.0
            preq_general_acc_s = 0.0

        # receive example
        xy = data[t, :]
        x = xy[:-1]
        x = np.reshape(x, (1, len(x)))  # reshape vector (d,) to 1-d matrix (1,d)
```

```
        # predict class label
        y_hat_prob = cls.model.predict(x=x, verbose=0)
        y_hat_class = np.around(y_hat_prob)

        # receive class label (ground truth)
        y = xy[-1]
        y = np.reshape(y, (1, 1))  # reshape from (1,) to (1,1)

        # update (train) classifier
        cls.train_cls(x, y)

        # check if prediction is correct
        correct = 1 if y == y_hat_class else 0

        # update accuracy
        preq_general_acc_s, preq_general_acc_n, preq_general_acc = \
            update_preq_metric(preq_general_acc_s, preq_general_acc_n, correct)
        preq_general_accs.append(preq_general_acc)

    # plot results
    plot_results(simulation_time, preq_general_accs, dataset_name, flag_store=flag_store_results)
```

4.3 Simulation experiments (stationary data)

Classifer parameters

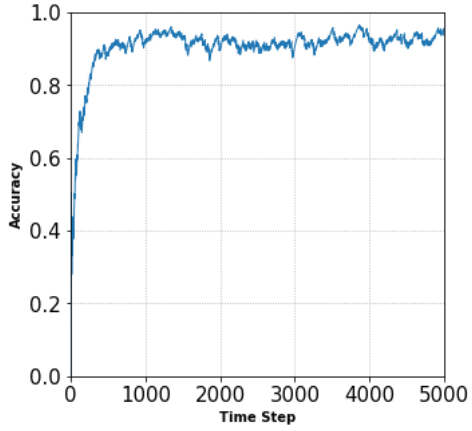
```
lr_params = {
    'cls_name': 'logistic_regression',
    'learning_rate': 0.01,
    'momentum': 0.9
}

nn_params = {
    'cls_name': 'neural_network',
    'learning_rate': 0.01,
    'momentum': 0.9,
    'hidden_units': 8,
    'hidden_activation_fun': 'tanh'
}
```

Simulation - Circle dataset

```
circle_params = {'dataset_name': 'circle', 'num_features': 2}
ogd(data_params=circle_params, cls_params=nn_params)
```

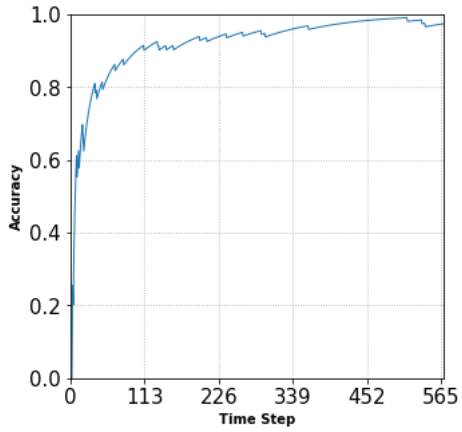
```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the le
Time step:  0
Time step:  500
Time step:  1000
Time step:  1500
Time step:  2000
Time step:  2500
Time step:  3000
Time step:  3500
Time step:  4000
Time step:  4500
```



Simulation - Breast Cancer dataset

```
cancer_params = {'dataset_name': 'cancer', 'num_features': 30}
ogd(data_params=cancer_params, cls_params=nn_params)
```

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the le
Time step:  0
Time step:  500
```

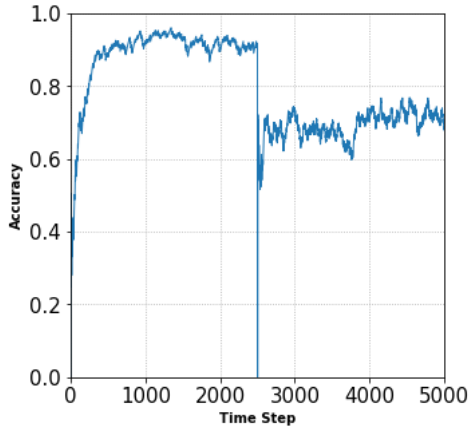


4.4 Simulation experiments (nonstationary data)

Simulation: Drifted circle dataset

```
circle_drift_params = {'dataset_name': 'circle_drift', 'num_features': 2}
ogd(data_params=circle_drift_params, cls_params=nn_params)
```

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the le
Time step:  0
Time step:  500
Time step:  1000
Time step:  1500
Time step:  2000
Time step:  2500
Time step:  3000
Time step:  3500
Time step:  4000
Time step:  4500
```



5. Prediction with Expert Advice

In this algorithm, a set of experts (i.e. an ensemble of classifiers) provide their advice on the prediction task.

5.1 Weighted Majority

"weighted_majority"

Algorithm

```
def weighted_majority(data_params, cls_params, num_classifiers=10, eta=0.5, t_drift=2500, flag_store_results=False):
    # get data parameters
    dataset_name = data_params['dataset_name']
    num_features = data_params['num_features']

    #  load data
    data = load_data(dataset_name)

    # init ensemble
    classifiers = [Classifier(num_features, cls_params, seed=i) for i in range(num_classifiers)]
    weights = [1.0 / num_classifiers] * num_classifiers

    # prequential evaluation
    preq_general_accs = []
    preq_general_acc_n = 0.0
    preq_general_acc_s = 0.0

    # start
    simulation_time = data.shape[0]
    for t in range(5000):

        if t % 500 == 0:
            print('Time step: ', t)

        if t == t_drift and 'drift' in dataset_name:
            preq_general_acc_n = 0.0
            preq_general_acc_s = 0.0

        # receive example
        xy = data[t, :]
        x = xy[:-1]
        x = np.reshape(x, (1, len(x)))  # reshape vector (d,) to 1-d matrix (1,d)

        # receive class label (ground truth)
        y = xy[-1]
        y = np.reshape(y, (1, 1))  # reshape from (1,) to (1,1)

        # experts_advice
        experts_advice_prob = []
        experts_advice_cost = []
        experts_advice_correct = []

        for cls in classifiers:
            y_hat_prob = cls.model.predict(x=x, verbose=0)
            y_hat_class = np.around(y_hat_prob)
            correct = 1 if y == y_hat_class else 0
            cost = 0 if y == y_hat_class else 1

            experts_advice_prob.append(y_hat_prob[0][0])
            experts_advice_correct.append(correct)
            experts_advice_cost.append(cost)

        # output class prediction (deterministic version)
        pred_class = np.around(np.average(experts_advice_prob, weights=weights))
        pred_class = np.array([pred_class])
        correct = 1 if y == np.reshape(pred_class, (1, 1)) else 0

        # update weights of classifiers
        weights = [weights[i] * np.exp(- eta * experts_advice_cost[i]) for i in range(num_classifiers)]
        weights = [w / sum(weights) for w in weights]  # normalise weights

        # update classifiers
        for cls in classifiers:
            cls.train_cls(x, y)

        # update accuracy
        preq_general_acc_s, preq_general_acc_n, preq_general_acc = \
            update_preq_metric(preq_general_acc_s, preq_general_acc_n, correct)
        preq_general_accs.append(preq_general_acc)

    # plot results
    plot_results(simulation_time, preq_general_accs, dataset_name, flag_store=flag_store_results)
```

Simulation: Drifted circle dataset

```
weighted_majority(data_params=circle_drift_params, cls_params=nn_params)
```

6. Links

Tutorials:

- Python 3
- NumPy
- Data pre-processing (scikit learn)