```
import numpy as np
import random
import warnings
from numpy.random import RandomState


class MAPEstimator():
    """
    Maximum A-Posteriori Estimator for musk probabilities

    Attributes
    ----------
    w_D   : D-dimensional vector of reals
            Defines weight vector
    prior : string
            Defines prior that will be used options: ('trivial','sas')
            'sas' stands for 'spike-and-slab'
    alpha : float, must be greater than 0
            Defines precision parameter of the multivariate Gaussian prior for the weight vector or the "spike" mul
    beta  : float, must be greater than 0 (optional)
            Defines precision parameter of the "slab" Gaussian on the GMM prior for the weight vector
    max_iter: integer
            Defines max number of iterations model can take to converge on fit()
    step_size: float must be greater than 0
            The step size of the gradient descent algorithm
    max_iter: int greater than 0
            Maximum number of iterations that the gradient descent algorithm will take


    Examples
    # TODO : Update this example or delete it
    --------
    >>> word_list = ['dinosaur', 'trex', 'dinosaur', 'stegosaurus']
    >>> mapEst = MAPEstimator(Vocabulary(word_list), alpha=2.0)
    >>> mapEst.fit(word_list)
    >>> np.allclose(mapEst.predict_proba('dinosaur'), 3.0/7.0)
    True

    >>> mapEst.predict_proba('never_seen-before')
    Traceback (most recent call last):
    ...
    KeyError: 'Word never_seen-before not in the vocabulary'
    """

    def __init__(self, w_D, prior='trivial', step_size_type = 'universal', alpha=1.0, beta=None, max_iter=30000, to
        #TODO decide on max_iter
        self.w_D = w_D
        self.c = 0
        self.alpha = float(alpha)
        self.max_iter = max_iter
        self.tol = tol
        self.step_size = step_size
        self.step_size_type = step_size_type
        self.iteration_count = iteration_count
        self.prior = prior
        self.loss_array = []
        if prior != 'trivial':
            self.beta = beta


    def fit(self, train_X, train_y):
        '''
        Fit this estimator to provided training data with first order stochastic gradient descent

        Args
        ----
        train_X : NxD array
            Each entry is a D-dimensional vector representing a training example
        train_y : Nx1 array
            Each entry is corresponding output class for training data


        Returns
        -------
        None. Internal attributes updated.

        Post Condition
```

```
        ---------------
        Attributes will updated based on provided word list
        * The 1D array count_V is set to the count of each vocabulary word
        * The integer total_count is set to the total length of the word list
        '''
        self.iteration_count = 0

        example_num = 0
        num_examples = len(train_X)

        self.loss_array = []

        if self.prior == 'trivial':

            count_10 = 0
            loss_10 = []
            L_avg = np.inf
            diff_L = np.inf
            L_avg_prev = np.inf

#             while(self.iteration_count <= self.max_iter) and (diff_L > self.tol):
#             # TODO : spit out a warning if the max_iter is reached
#                 #print('iteration: %i' % self.iteration_count)

#                 h_x = np.matmul(self.w_D.transpose(), train_X) + self.c
#                 #print('h_x: ' + str(h_x))
# #                 print('w_D: ' + str(self.w_D))
# #                 print('c: ' + str(self.c))
#                 sig = 1 / (1 + np.exp(-h_x))
# #                 print('sig: ' + str(sig))
# #                 print('train_y: ' + str(train_y[example_num]))
#                 L = np.dot(train_y, np.log(sig)) + np.dot((1-train_y), np.log(1-sig))
#                 print('L:' + str(L))
# #                 loss_10.append(L)
# #                 count_10 += 1

# #                 if count_10 == 1000:
#                 diff_L = np.abs(L_avg_prev - L)
#                 L_avg_prev = L

#                 self.w_D = self.w_D - self.step_size * (np.matmul(train_X.transpose(),(sig - train_y)) + self.alp
#                 self.c = self.c - self.step_size * (sig - train_y)

# #                 if example_num >= num_examples:
# #                     example_num = 0
# #                 self.iteration_count += 1

#             if self.iteration_count == self.max_iter:
#                 warnings.warn("Maximum iterations reached")

            prng = RandomState(136)

            while(self.iteration_count <= self.max_iter) and (diff_L > self.tol):
            # TODO : spit out a warning if the max_iter is reached
                #print('iteration: %i' % self.iteration_count)
                example_num = prng.randint(0, num_examples-1)
                h_x = np.dot(self.w_D, train_X[example_num]) + self.c
                #print('h_x: ' + str(h_x))
#                 print('w_D: ' + str(self.w_D))
#                 print('c: ' + str(self.c))
                sig = 1 / (1 + np.exp(-h_x))
#                 print('sig: ' + str(sig))
#                 print('train_y: ' + str(train_y[example_num]))
                L = train_y[example_num] * np.log(sig) + (1-train_y[example_num]) * np.log(1-sig)
                #print('L:' + str(L))
                loss_10.append(L)
                count_10 += 1

                if count_10 == 100000:
                    L_avg = np.mean(loss_10)
#                     print(L_avg)
                    self.loss_array.append(L_avg)
                    diff_L = np.abs(L_avg_prev - L_avg)
                    count_10 = 0
                    loss_10 = []
                    L_avg_prev = L_avg
```

```
                if self.step_size_type == 'universal':
                    self.w_D = self.w_D - self.step_size * ((sig - train_y[example_num]) * train_X[example_num] + s
                    self.c = self.c - self.step_size * (sig - train_y[example_num])
                else:
                    self.w_D = self.w_D - (self.step_size + train_y[example_num]*0.1)* ((sig - train_y[example_num]
                    self.c = self.c - (self.step_size + train_y[example_num]*0.1) * (sig - train_y[example_num])

#                 example_num += 1
#                 if example_num >= num_examples:
#                     example_num = 0
                self.iteration_count += 1

            if self.iteration_count == self.max_iter:
                warnings.warn("Maximum iterations reached")

        #else:
            #TODO: Code up the SGD for the sas prior

    def predict_proba(self, test_X):
        '''
        Predict probability of a given set of feature vectors under this model

        Args
        ----
        test_X : NxD vector
            Examples for which probability will be predicted

        Returns
        -------
        proba : float between 0 and 1

        Throws
        ------
        ValueError if hyperparameters do not allow MAP estimation
        KeyError if the provided word is not in the vocabulary
        '''

        lin_preds = np.matmul(test_X, self.w_D)

        sig_preds = 1 / (1 + np.exp(-lin_preds))

        return sig_preds

    def score(self, test_X, test_y):
        ''' Compute the average log probability of words in provided list

        Args
        ----
        test_X : NxD array
            Each entry is a D-dimensional vector representing a test example
        test_y : Nx1 array
            Each entry is corresponding output class for test data

        Returns
        -------
        avg_log_proba : float between (-np.inf, 0.0)
        '''
        correct_count = 0
        num_examples = len(test_X)

        pred_proba = self.predict_proba(test_X)
        pred_class = pred_proba > 0.5
        is_correct = pred_class == test_y

        return np.sum(is_correct) / num_examples
```

test_gd

April 17, 2022

[1]:
```
## Abstractions
import numpy as np
import pandas as pd

## Plotting
from matplotlib import pyplot as plt
plt.style.use('seaborn')
import seaborn as sns
import pylab as pl

## Scalers
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

## Models
from sklearn.linear_model import LogisticRegression

## Model Selection
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold

## Timing
import time

# Model
from MAPEstimator import MAPEstimator
```

Import Data

[2]:
```
headers = ['molecule_name', 'conformation_name']
for i in range(1, 167):
    name = 'f%i' % i
    headers.append(name)
headers.append('class')
```

[3]:
```
# headers = pd.read_csv('clean2.info')
df = pd.read_csv('src/clean2.data')
```

```
df.columns = headers
```

[4]:
```
X = np.asarray(df.iloc[:,2:-1])
y = np.asarray(df.iloc[:,-1])
```

Standard Scaler

[5]:
```
X_std = StandardScaler().fit_transform(X)
```

Train and Test Model

[6]:
```
clf = MAPEstimator(w_D = np.zeros(X.shape[1]), step_size=0.1, alpha=0.1,␣
 ↪max_iter = 10000000)
clf.fit(X_std,y)
predict_y = clf.predict_proba(X_std)
```

[7]:
```
score = clf.score(X_std, y)
score
```

[7]: 0.6296801576474155

[8]:
```
(np.sum(y==0)) / len(y)
```

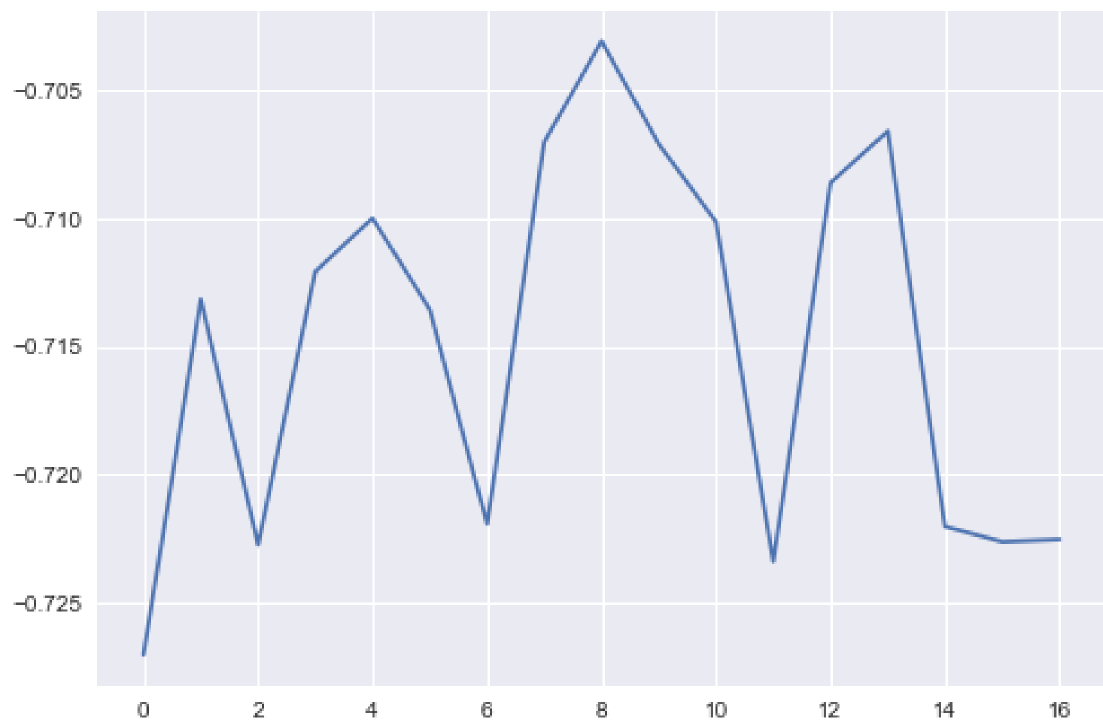
[8]: 0.8459906017886918

[10]:
```
clf.iteration_count
```

[10]: 1700000

[11]:
```
plt.plot(clf.loss_array)
```

[11]: [<matplotlib.lines.Line2D at 0x7fd71c99fee0>]



[17]:
```
clf = MAPEstimator(w_D = np.zeros(X.shape[1]), step_size=0.1, alpha=0.1,␣
 ↪max_iter = 10000000, step_size_type = 'differential')
clf.fit(X_std,y)
predict_y = clf.predict_proba(X_std)
```

[18]:
```
score = clf.score(X_std, y)
score
```

[18]: 0.6396847051690162

[19]:
```
(np.sum(y==0)) / len(y)
```

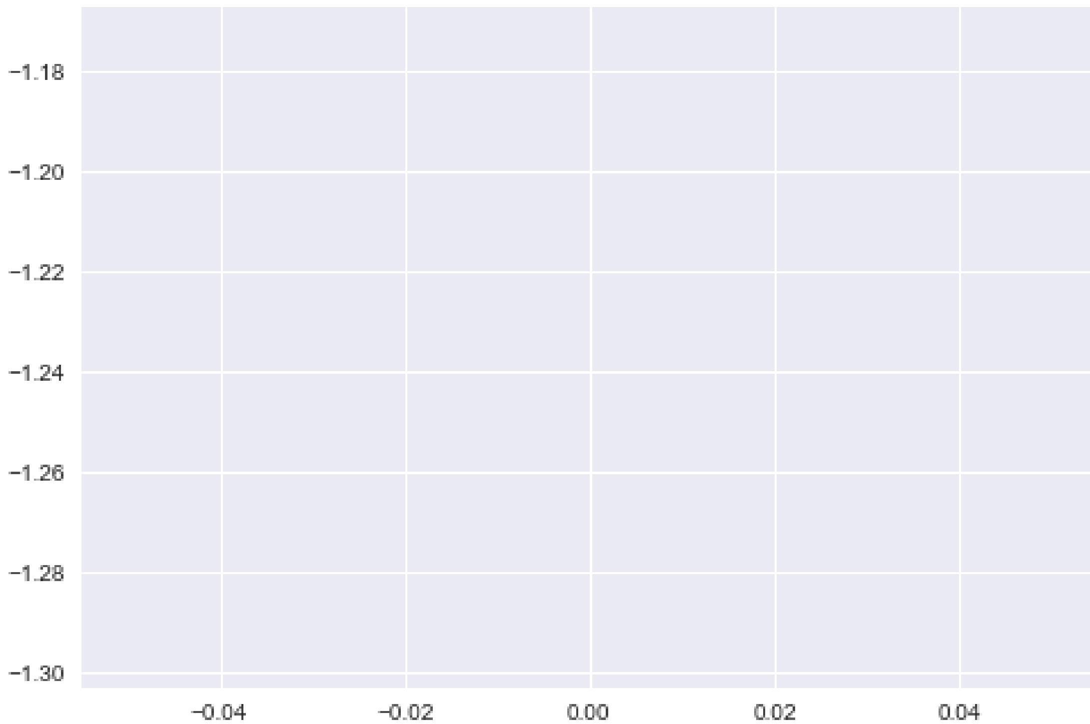
[19]: 0.8459906017886918

[20]:
```
clf.iteration_count
```

[20]: 300000

[21]:
```
plt.plot(clf.loss_array)
```

[21]: [<matplotlib.lines.Line2D at 0x7fd71cb516a0>]



Model Evaluation

[6]:
```
kf = KFold(n_splits=10, shuffle = True, random_state = 136)
kf.get_n_splits(X_std, y)
```

[6]: 10

[7]:
```
iteration_counts = []
test_scores = []

for train_index, test_index in kf.split(X_std, y):
    X_train, X_test = X_std[train_index], X_std[test_index]
    y_train, y_test = y[train_index], y[test_index]
    clf = MAPEstimator(w_D = np.zeros(X.shape[1]), step_size=0.1, alpha=0.1,␣
 ↪max_iter = 10000000)
    clf.fit(X_train,y_train)
    iteration_counts.append(clf.iteration_count)
    score = clf.score(X_test, y_test)
    test_scores.append(score)
```

[8]:
```
test_scores
```

[8]:

```
[0.6181818181818182,
 0.5409090909090909,
 0.6151515151515151,
 0.6439393939393939,
 0.5772727272727273,
 0.6,
 0.5590909090909091,
 0.575113808801214,
 0.6236722306525038,
 0.6585735963581184]
```

[9]:

```
iteration_counts
```

[9]:

```
[470000,
 2986000,
 155000,
 150000,
 1942000,
 4821000,
 352000,
 507000,
 1327000,
 260000]
```

[10]:

```
basic_test_scores = test_scores
```

[11]:

```
basic_iteration_counts = iteration_counts
```

[12]:

```
np.mean(basic_test_scores)
```

[12]:

```
0.6011905090357291
```

[13]:

```
np.mean(basic_iteration_counts)
```

[13]:

```
1297000.0
```

[14]:

```
kf = KFold(n_splits=10, shuffle = True, random_state = 136)
kf.get_n_splits(X_std, y)
```

[14]:

```
10
```

[15]:

```
iteration_counts = []
test_scores = []

for train_index, test_index in kf.split(X_std, y):
    X_train, X_test = X_std[train_index], X_std[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
    clf = MAPEstimator(w_D = np.zeros(X.shape[1]), step_size=0.1, alpha=0.1,␣
↪max_iter = 10000000, step_size_type = 'differential')
    clf.fit(X_train,y_train)
    iteration_counts.append(clf.iteration_count)
    score = clf.score(X_test, y_test)
    test_scores.append(score)
```

```
/Users/nathanieldavis/Documents/tufts/2022spring/cs136/project/cs136_final_proje
ct/Checkpoint 2/MAPEstimator.py:141: RuntimeWarning: divide by zero encountered
in log
  L = train_y[example_num] * np.log(sig) + (1-train_y[example_num]) *
np.log(1-sig)
/Users/nathanieldavis/Documents/tufts/2022spring/cs136/project/cs136_final_proje
ct/Checkpoint 2/MAPEstimator.py:141: RuntimeWarning: invalid value encountered
in double_scalars
  L = train_y[example_num] * np.log(sig) + (1-train_y[example_num]) *
np.log(1-sig)
```

[16]:
```
test_scores
```

[16]:
```
[0.6393939393939394,
 0.6287878787878788,
 0.6272727272727273,
 0.696969696969697,
 0.6681818181818182,
 0.6106060606060606,
 0.5015151515151515,
 0.5386949924127465,
 0.5766312594840668,
 0.6176024279210925]
```

[17]:
```
iteration_counts
```

[17]:
```
[273000,
 889000,
 1997000,
 272000,
 301000,
 411000,
 1267000,
 2137000,
 434000,
 751000]
```

[18]:
```
differential_test_scores = test_scores
```

[19]:
```
differential_iteration_counts = iteration_counts
```

[20]:
```
np.mean(differential_test_scores)
```

[20]: 0.6105655952545179

[21]:
```
np.mean(differential_iteration_counts)
```

[21]: 873200.0

[22]:
```
plot_test_scores = np.array([np.mean(basic_test_scores),np.
 ↪mean(differential_test_scores)])
```

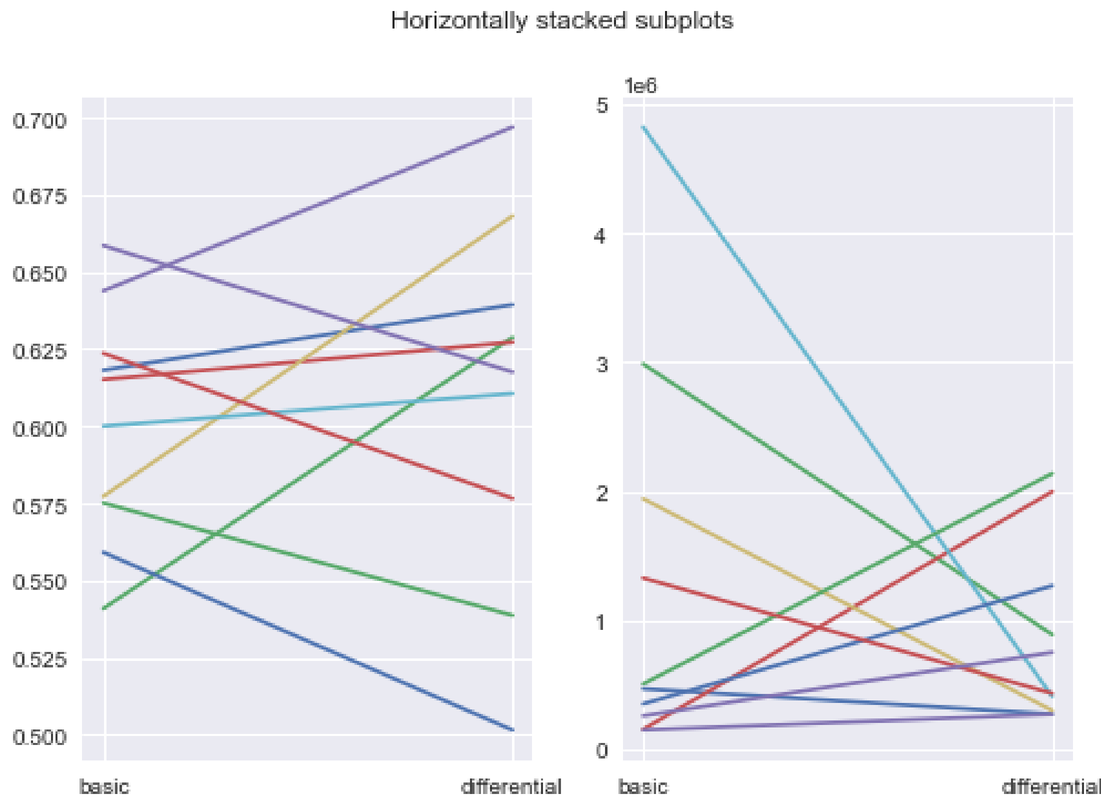
[23]:
```
plot_iteration_counts = np.array([np.mean(basic_iteration_counts), np.
 ↪mean(differential_iteration_counts)])
```

[24]:
```
plot_test_scores = np.array([basic_test_scores, differential_test_scores])
```

[25]:
```
plot_iteration_counts = np.array([basic_iteration_counts,␣
 ↪differential_iteration_counts])
```

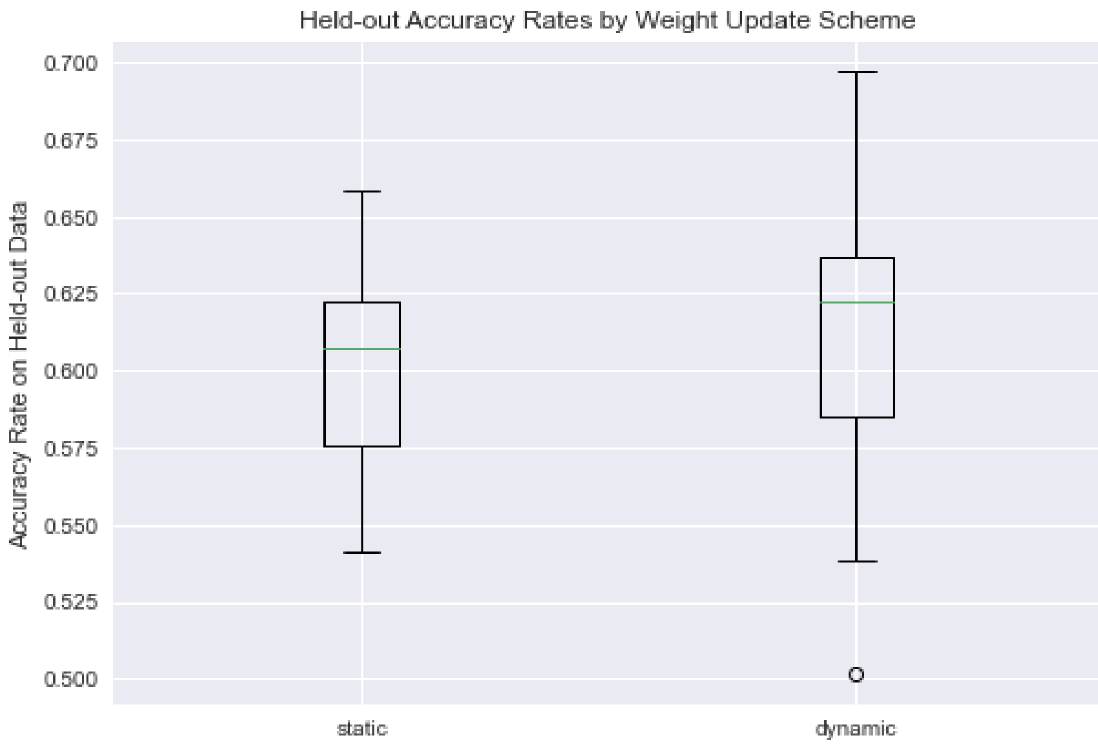
[26]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Horizontally stacked subplots')
ax1.plot(['basic','differential'],plot_test_scores)
ax2.plot(['basic','differential'], plot_iteration_counts)
```

[26]:
```
[<matplotlib.lines.Line2D at 0x7f9bf0f40c10>,
 <matplotlib.lines.Line2D at 0x7f9bf0f40bb0>,
 <matplotlib.lines.Line2D at 0x7f9bf0f40d60>,
 <matplotlib.lines.Line2D at 0x7f9bf0f40e80>,
 <matplotlib.lines.Line2D at 0x7f9bf0f40fa0>,
 <matplotlib.lines.Line2D at 0x7f9bf0f4e100>,
 <matplotlib.lines.Line2D at 0x7f9bf0f40c40>,
 <matplotlib.lines.Line2D at 0x7f9bf0f4e220>,
 <matplotlib.lines.Line2D at 0x7f9bf0f4e430>,
 <matplotlib.lines.Line2D at 0x7f9bf0f4e550>]
```



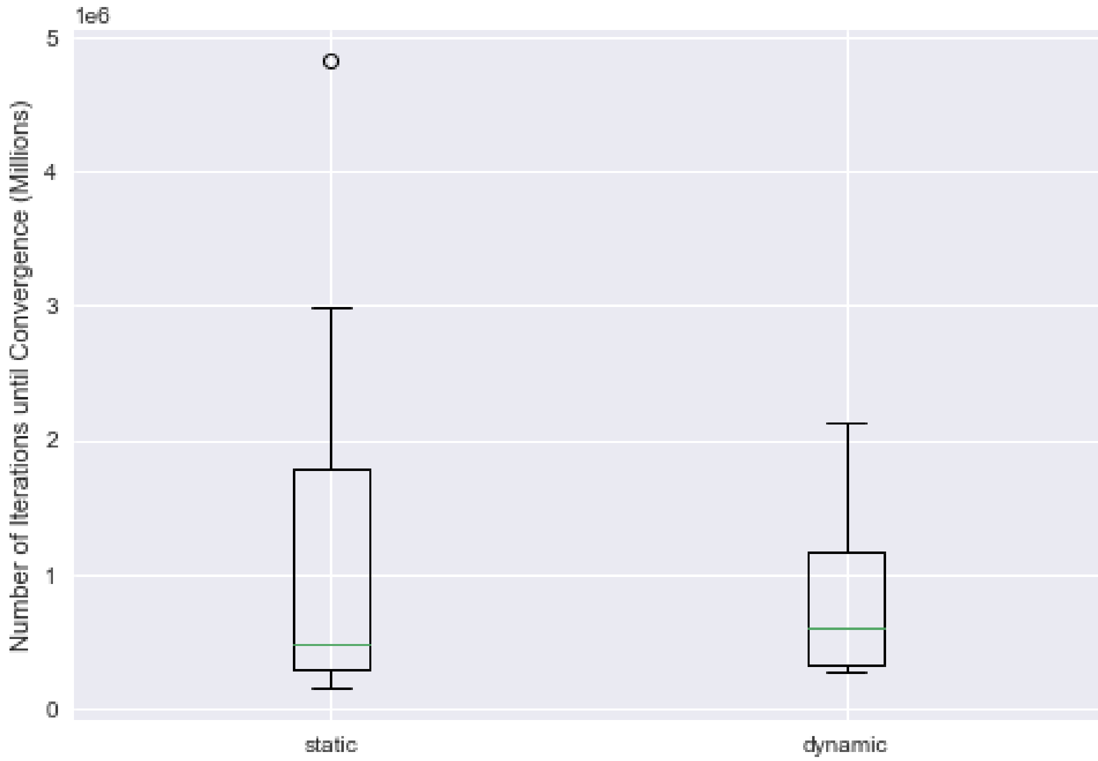

```
[35]: plt.boxplot(np.transpose(plot_test_scores), labels = ['static', 'dynamic'])
      plt.title('Held-out Accuracy Rates by Weight Update Scheme')
      plt.ylabel('Accuracy Rate on Held-out Data')
```

```
[35]: Text(0, 0.5, 'Accuracy Rate on Held-out Data')
```



[36]:
```
plt.boxplot(np.transpose(plot_iteration_counts), labels = ['static', 'dynamic'])
plt.ylabel('Number of Iterations until Convergence (Millions)')
```

[36]: Text(0, 0.5, 'Number of Iterations until Convergence (Millions)')



[51]:
```
plot_test_scores
```

[51]:
```
array([[0.61818182, 0.63787879, 0.61515152, 0.64393939, 0.59848485,
        0.5530303 , 0.55909091, 0.57511381, 0.53717754, 0.6585736 ],
       [0.63939394, 0.62878788, 0.63484848, 0.6969697 , 0.66818182,
        0.61060606, 0.65151515, 0.61001517, 0.57663126, 0.61760243]])
```

[39]:
```
plt.plot(clf.loss_array)
```

[39]:
```
[<matplotlib.lines.Line2D at 0x7f9bf15eed60>]
```

−0.8
−1.0
−1.2
−1.4
−1.6
0
100
200
300
400
500
600
700


```
[ ]:
```