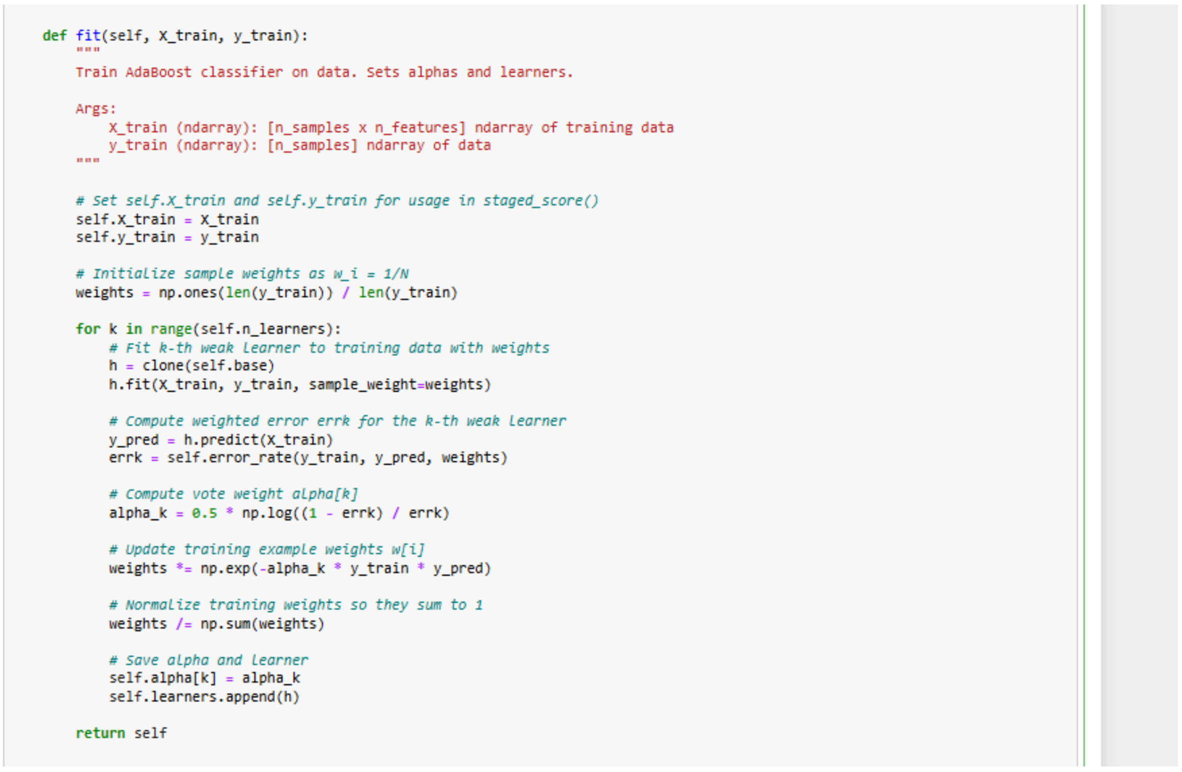
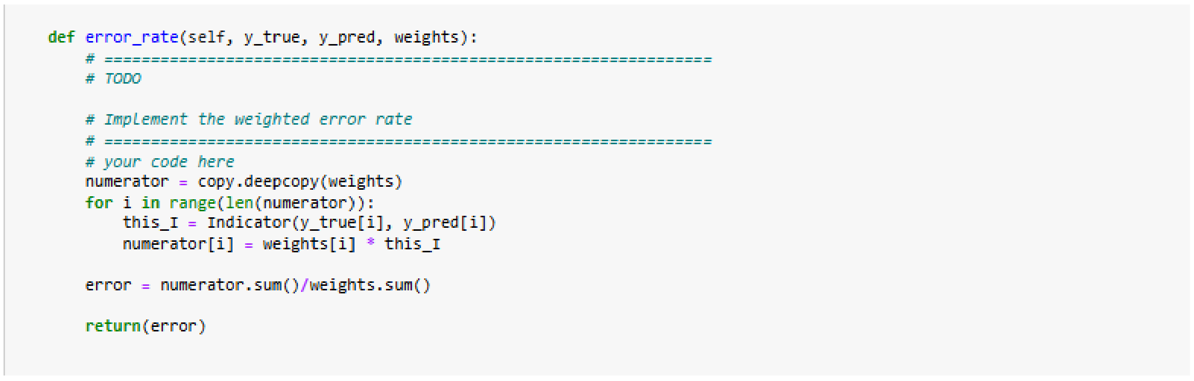
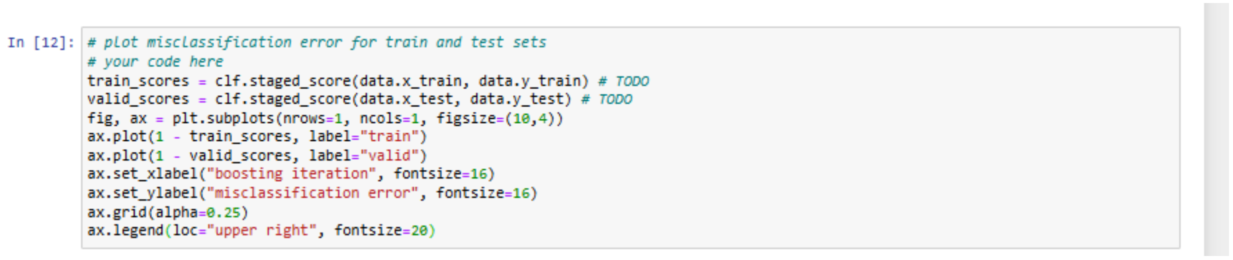
Week 5 Peer Review



```
def fit(self, X_train, y_train):
    """
    Train AdaBoost classifier on data. Sets alphas and learners.

    Args:
        X_train (ndarray): [n_samples x n_features] ndarray of training data
        y_train (ndarray): [n_samples] ndarray of data
    """

    # Set self.X_train and self.y_train for usage in staged_score()
    self.X_train = X_train
    self.y_train = y_train

    # Initialize sample weights as w_i = 1/N
    weights = np.ones(len(y_train)) / len(y_train)

    for k in range(self.n_learners):
        # Fit k-th weak learner to training data with weights
        h = clone(self.base)
        h.fit(X_train, y_train, sample_weight=weights)

        # Compute weighted error errk for the k-th weak learner
        y_pred = h.predict(X_train)
        errk = self.error_rate(y_train, y_pred, weights)

        # Compute vote weight alpha[k]
        alpha_k = 0.5 * np.log((1 - errk) / errk)

        # Update training example weights w[i]
        weights *= np.exp(-alpha_k * y_train * y_pred)

        # Normalize training weights so they sum to 1
        weights /= np.sum(weights)

        # Save alpha and learner
        self.alpha[k] = alpha_k
        self.learners.append(h)

    return self
```



```
def error_rate(self, y_true, y_pred, weights):
    # =================================================================
    # TODO

    # Implement the weighted error rate
    # =================================================================
    # your code here
    numerator = copy.deepcopy(weights)
    for i in range(len(numerator)):
        this_I = Indicator(y_true[i], y_pred[i])
        numerator[i] = weights[i] * this_I

    error = numerator.sum()/weights.sum()

    return(error)
```

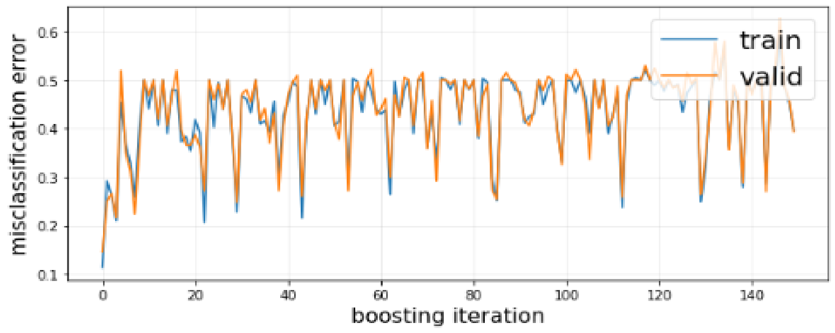


```
In [12]: # plot misclassification error for train and test sets
         # your code here
         train_scores = clf.staged_score(data.X_train, data.y_train) # TODO
         valid_scores = clf.staged_score(data.X_test, data.y_test) # TODO
         fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,4))
         ax.plot(1 - train_scores, label="train")
         ax.plot(1 - valid_scores, label="valid")
         ax.set_xlabel("boosting iteration", fontsize=16)
         ax.set_ylabel("misclassification error", fontsize=16)
         ax.grid(alpha=0.25)
         ax.legend(loc="upper right", fontsize=20)
```

```
    def create_tree(self,i):
        """
        create a single decision tree classifier
        """

        idxs = np.random.permutation(len(self.y))[:self.sample_sz]
        idxs = np.asarray(idxs)

        f_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
        f_idxs = np.asarray(f_idxs)


        if i==0:
            self.features_set = np.array(f_idxs, ndmin=2)
        else:
            self.features_set = np.append(self.features_set, np.array(f_idxs,ndmin=2),axis=0)

        # TODO: build a decision tree classifier and train it with x and y that is a subset of data (use idxs and f_idxs)

        # your code here
        clf = DecisionTreeClassifier(max_depth = self.max_depth, min_samples_leaf = self.min_samples_leaf)
        this_x = self.x[idxs, :]
        this_x = this_x[:, f_idxs]
        this_y = self.y[idxs]
        clf.fit(this_x, this_y)
```

```
    def predict(self, x):

        # TODO: create a vector of predictions  and return
        # You will have to return the predictions of the final ensembles based on the individual trees' predicitons


        # your code here
        to_return = np.zeros(x.shape[0])

        for i in range(x.shape[0]):
            #Get all individual predictions for x[i, :].reshape(1, -1) from trees, aggregate into dictionary
            all_predictions = dict()
            for j in range(len(self.trees)):
                this_tree = self.trees[j]
                these_features = self.features_set[j]
                this_prediction = this_tree.predict(x[i, these_features].reshape(1, -1))[0]
                if this_prediction not in all_predictions.keys():
                    all_predictions.update({this_prediction : 1})
                else:
                    previous_count = all_predictions[this_prediction]
                    all_predictions.update({this_prediction : previous_count + 1})

            #Decide ensemble prediction for x[i, :].reshape(1, -1) by majority vote
            current_best = (None, 0)
            for key, value in all_predictions.items():
                if value > current_best[1]:
                    current_best = (key, value)

            to_return[i] = current_best[0]

        return(to_return)
```

```
Out[12]: <matplotlib.legend.Legend at 0x70f8d2cd2d50>
```



In [17]:

```
# TODO: build a random forest classifier and make predictions

# your code here
sizes = [100 * i for i in range(1,11)]
errors = []
for j in range(len(sizes)):
    this_Forest = RandomForest(data.x_train, data.y_train, sample_sz = sizes[j])
    this_score = this_Forest.score(data.x_test, data.y_test)
    errors.append(1 - this_score)
print(errors)
```

```
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
21 sha:  441
[0.07799999999999996, 0.07599999999999996, 0.06999999999999995, 0.06799999999999995, 0.05800000000000005, 0.05200000000000046,
0.04600000000000004, 0.05200000000000046, 0.05000000000000044, 0.04200000000000004]
```