LINEAR SVM

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as io
import libsvm
from libsvm.svmutil import *

%matplotlib inline
```

3.1 Linear Support Vector Machine on toy data

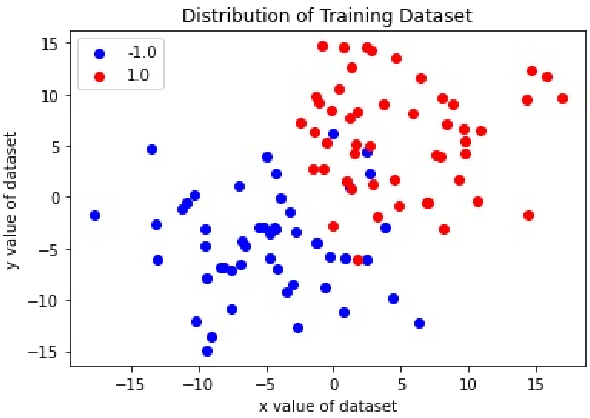
3.1.1

Generate a training set of size 100 with 2D features (X) drawn at random as follows:

- X_{neg} $\sim$ $\mathcal{N}$([-5, -5], 5*I_2) and correspond to negative labels (-1)
- X_{pos} $\sim$ $\mathcal{N}$([5, 5], 5*I_2) and correspond to positive labels (+1)
  Accordingly, $X = [X_{neg}, X_{pos}]$ is a 100×2 array, Y is a 100×1 array of values $\in \{-1, 1\}$.
  Draw a scatter plot of the full training dataset with the points colored according to their labels.

In [135…

```
# Generate binary class dataset
np.random.seed(0)

n_samples = 100
center_1 = [-5, -5]
center_2 = [5, 5]

# Generate Data:
Xneg = np.random.normal(center_1,5,size=(50,2))
Xpos = np.random.normal(center_2,5,size=(50,2))
X = np.concatenate((Xneg, Xpos), axis=0)
#Y = np.array([-1]*50,[1]*50)
#Y = np.reshape(Y,(100,1))
np.concatenate((-1*np.ones(50),np.ones(50)), axis=0)

#print(Xneg)
#print(Xpos)

# Scatter plot:
fig, ax = plt.subplots()
plt.title("Distribution of Training Dataset")
plt.xlabel("x value of dataset")
plt.ylabel("y value of dataset")
cdict = {-1: 'blue', 1: 'red'}
unique_Y = np.unique(Y)
for g in unique_Y:
    wx = np.where(Y == g)
    ax.scatter(X[wx,0], X[wx,1], c = cdict[g], label = g)
    ax.legend()
plt.show()
```



3.1.2

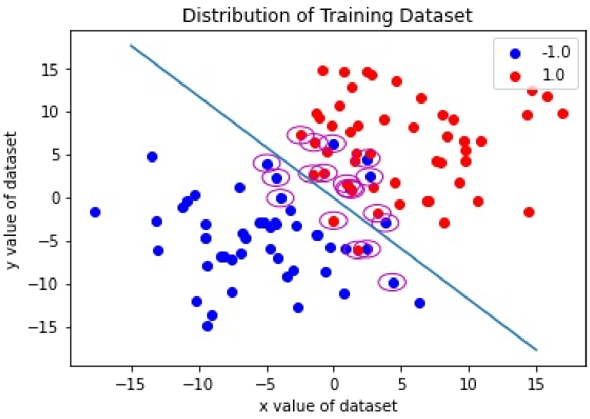
Train a linear support vector machine on the data with $C = 1$ and draw the decision boundary line that separates o and x. Mark the support vectors separately (ex.circle around the point).

Note: You can use the libsvm.svmutil functions with the kernel_type set to 0, indiciating a linear kernel and svm_type set to 0 indicating C-SVC. Also note that the support_vector coefficients returned by the LIBSVM model are the dual coefficients.

In [139…

```
# Define the SVM problem
prob = svm_problem(Y,X)

# Define the hyperparameters
param = svm_parameter('-s 0 -t 0 -c 1')

# Train the model
model = svm_train(prob, param)

# Compute the slope and intercept of the separating line/hyperplanee with the use of the support vectors
# and other information from the LIBSVM model.
p_labs, p_acc, p_vals = svm_predict(Y, X, model)

w = np.matmul(X[np.array(model.get_sv_indices()) - 1].T, model.get_sv_coef())
b = -model.rho.contents.value

# Draw the scatter plot, the decision boundary line, and mark the support vectors.

space_x = np.linspace(-15, 15, 100)

cdict = {-1: 'blue', 1: 'red'}
fig, ax = plt.subplots()
plt.title("Distribution of Training Dataset")
plt.xlabel("x value of dataset")
plt.ylabel("y value of dataset")

for g in np.unique(Y):
    wx = np.where(Y == g)
    ax.scatter(X[wx,0], X[wx,1], c = cdict[g], label = g)

plt.plot(dummy_x, (-w[0]/w[1])*dummy_x-(b/w[1]))

for i in model.get_sv_indices():
    circle = plt.Circle((X[i-1,0], X[i-1,1]),1,color='m', fill = False)
    ax.add_patch(circle)


ax.legend()
plt.show()
```

```
Accuracy = 93% (93/100) (classification)
```



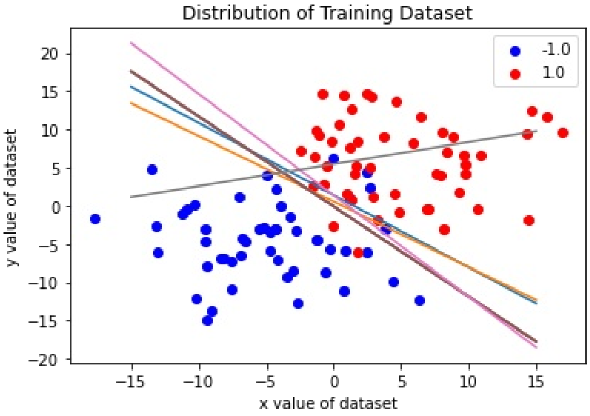
3.1.3

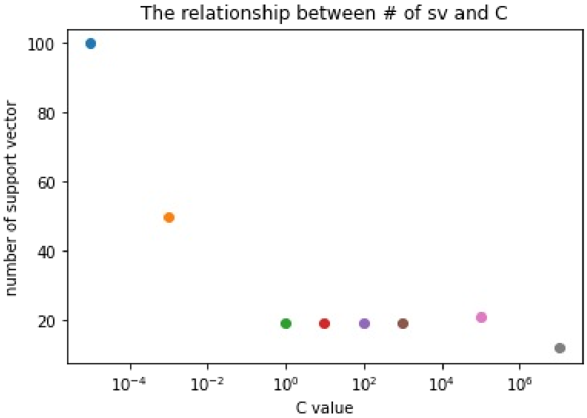
Draw a line that separates the data for 8 different C (10^{-5}~10^{7}). Plot the number of support vectors vs. C (plot x-axis on a log scale). How does the number of support vectors change as C increases and why does it change like that?

Note: You might prefer to use the command-line style of svm_parameter initialization such as: svm_parameter('-s 0 -t 0') to indicate a linear kernel and C-SVC as the SVM type.

In [155…

```
C_range = [10**-5, 10**-3, 1, 10, 100, 10**3, 10**5, 10**7]
num_sv = []
coefficient_data = []

# Loop over a similar setup to that in the previous code block.
for i in range(len(C_range)):
    prob = svm_problem(Y,X)
    param = svm_parameter('-s 0 -t 0 -c {}'.format(C_range[i]))
    model = svm_train(prob, param)
    rho=model.rho[0]
    w = np.matmul(X[np.array(model.get_sv_indices()) - 1].T, model.get_sv_coef())
    b = -model.rho.contents.value

    num_sv.append(len(model.get_sv_indices()))
    coefficient_data.append(((-w[0]/w[1]),-(b/w[1])))

# Draw the scatter plot with multiple decision lines on top (one for each value of C)
space_x = np.linspace(-15, 15, 100)

cdict = {-1: 'blue', 1: 'red'}
fig, ax = plt.subplots()
plt.title("Distribution of Training Dataset")
plt.xlabel("x value of dataset")
plt.ylabel("y value of dataset")

for g in np.unique(Y):
    wx = np.where(Y == g)
    ax.scatter(X[wx,0], X[wx,1], c = cdict[g], label = g)

for i in range(len(coefficient_data)):
    plt.plot(space_x, coefficient_data[i][0]*space_x+coefficient_data[i][1])

ax.legend()
plt.show()

# Draw the num_sv vs. C plot.

fig, ax = plt.subplots()
plt.title("The relationship between # of sv and C")
plt.xlabel("C value")
plt.ylabel("number of support vector")
for i in range(len(C_range)):
    ax.set_xscale('log')
    ax.scatter(C_range[i], num_sv[i])

plt.show()
```





The number of support vectors decreses as C value increases. If C value is increased, there should be a greater penalty of the constraint. The size of violations is reduced, and it causes the margin will get narrower. Therefore, there are fewer support vectors by increasing C value.

3.1.4

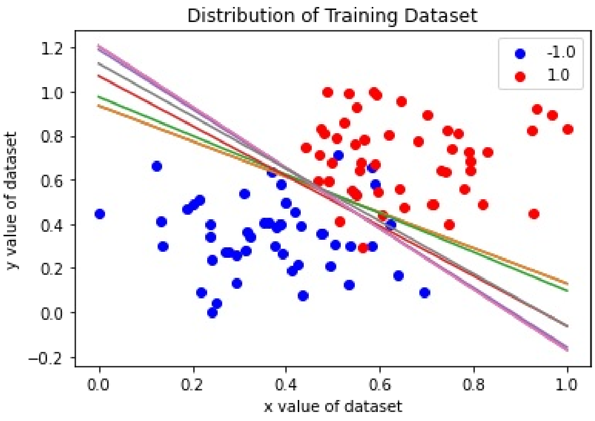
Now try rescaling the data to the [0,1] range and repeat the steps of the previous question (3.1.3) and over the same range of C values. Are the decision boundaries different from those in the previous question? What does this imply about (a) the geometric margin and (b) the relative effect of each feature on the predictions of the trained model ?

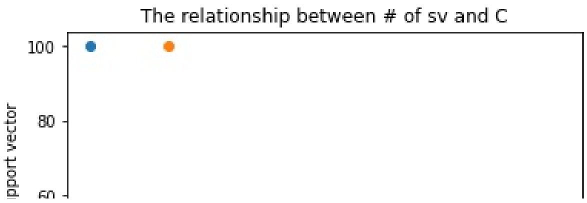
In [157…

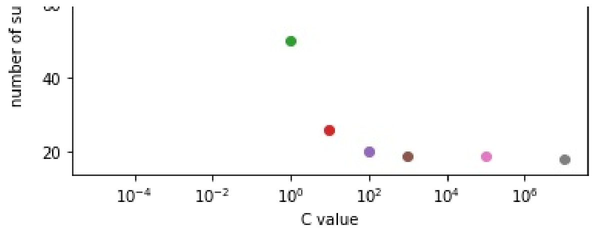
```
import sklearn
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler()

# Single line below:
X_train_minmax = min_max_scaler.fit_transform(X)
```

In [160…

```
C_range = [10**-5, 10**-3, 1, 10, 100, 10**3, 10**5, 10**7]
num_sv = []
coefficient_data = []

# Repeat the loop from 3.1.3

# Loop over a similar setup to that in the previous code block.
for i in range(len(C_range)):
    prob = svm_problem(Y,X_train_minmax)
    param = svm_parameter('-s 0 -t 0 -c {}'.format(C_range[i]))
    model = svm_train(prob, param)

    rho=model.rho[0]
    w = np.matmul(X_train_minmax[np.array(model.get_sv_indices()) - 1].T, model.get_sv_coef())
    b = -model.rho.contents.value

    num_sv.append(len(model.get_sv_indices()))
    coefficient_data.append(((-w[0]/w[1]),-(b/w[1])))

# Draw the scatter plot with multiple decision lines on top (one for each value of C)
space_x = np.linspace(0, 1, 100)

cdict = {-1: 'blue', 1: 'red'}
fig, ax = plt.subplots()
plt.title("Distribution of Training Dataset")
plt.xlabel("x value of dataset")
plt.ylabel("y value of dataset")

for g in np.unique(Y):
    wx = np.where(Y == g)
    ax.scatter(X_train_minmax[wx,0], X_train_minmax[wx,1], c = cdict[g], label = g)

for i in range(len(coefficient_data)):
    plt.plot(space_x, coefficient_data[i][0]*space_x+coefficient_data[i][1])

ax.legend()
plt.show()

# Draw the num_sv vs. C plot.

fig, ax = plt.subplots()
plt.title("The relationship between # of sv and C")
plt.xlabel("C value")
plt.ylabel("number of support vector")
for i in range(len(C_range)):
    ax.set_xscale('log')
    ax.scatter(C_range[i], num_sv[i])

plt.show()
```







The decision boundaries for this section are different from those on the previous section. It is because the boundary of x value is different. In this problem, the range [0,1] has been used unlike the range of [-15,15] for the last section. Geometrically, the margins between points are smaller, it makes the prediction harder to decide than the previeous geometrical margin.

Penguins

```
# This is formatted as code
```

Multiclass SVM. In this problem, we will use support vector machines to classify penguins species based on the given features.

Load in the penguins data using from the provided penguins.csv file on Sakai. Preprocess the features. Then we will use the train_test_split from sklearn to split the data into training and testing into a 0.8 : 0.2 ratio.

In [172…

```
import pandas as pd
from sklearn.model_selection import train_test_split

# load data
df = pd.read_csv('penguins.csv')
print(df.head())

# split data
X_train, X_test, y_train, y_test = train_test_split(df.drop(['Species'], axis=1).to_numpy(), df['Species'].to_num
```

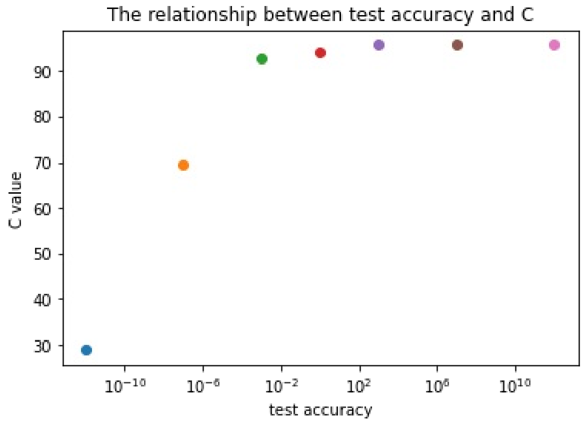
```
   CulmenLength  CulmenDepth  FlipperLength  BodyMass  Species
0          39.1         18.7          181.0    3750.0        0
1          39.5         17.4          186.0    3800.0        0
2          40.3         18.0          195.0    3250.0        0
3           NaN          NaN            NaN       NaN        0
4          36.7         19.3          193.0    3450.0        0
```

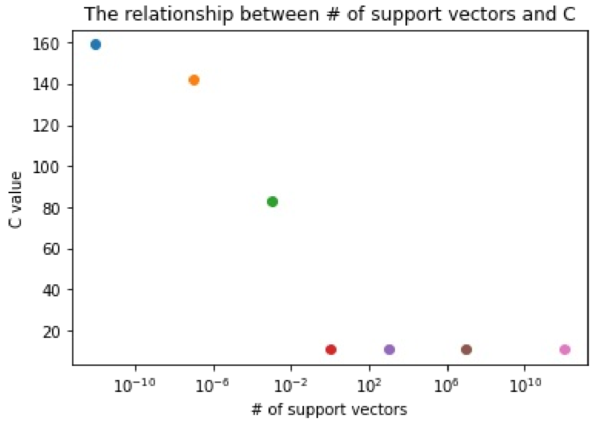
Train the support vector machine classifier with a linear kernel on the first 5000 datapoints and test the accuracy on the following 5000 points. Plot test accuracy and the number of support vectors (two separate plots) vs. C for $C = 10^{-12}$~10^{12} (plot 7 points or more with the x-axis on a log scale).

In [173…

```
C_range = [10**-12, 10**-7, 10**-3, 1, 10**3, 10**7, 10**12]

num_sv = []
ans = []
accuracy = []

# Loop over a similar setup to that in the previous code block.
for i in range(len(C_range)):
    prob = svm_problem(y_train,X_train)
    param = svm_parameter('-s 0 -t 0 -c {}'.format(C_range[i]))
    model = svm_train(prob, param)

    rho=model.rho[0]
    w = np.matmul(X_train[np.array(model.get_sv_indices()) - 1].T, model.get_sv_coef())
    b = -model.rho.contents.value

    p_labels, p_acc, p_vals = svm_predict(y_test, X_test, model)
    num_sv.append(len(model.get_sv_indices()))
    accuracy.append(p_acc)

for i in range(len(accuracy)):
    ans.append(accuracy[i][0])

# Draw the scatter plot with multiple decision lines on top (one for each value of C)
space_x = np.linspace(-15, 15, 100)

fig, ax = plt.subplots()
plt.title("The relationship between test accuracy and C")
plt.xlabel("test accuracy")
plt.ylabel("C value")
for i in range(len(C_range)):
    ax.scatter(C_range[i], ans[i])
    ax.set_xscale('log')
plt.show

# Draw the num_sv vs. C plot.
fig, ax = plt.subplots()
plt.title("The relationship between # of support vectors and C")
plt.xlabel("# of support vectors")
plt.ylabel("C value")
for i in range(len(C_range)):
    ax.scatter(C_range[i], num_sv[i])
    ax.set_xscale('log')
plt.show()
```

```
Accuracy = 28.9855% (20/69) (classification)
Accuracy = 69.5652% (48/69) (classification)
Accuracy = 92.7536% (64/69) (classification)
Accuracy = 94.2029% (65/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
```





Calculate variable importance (CulmenLength, CulmenDepth, FlipperLength, BodyMass) with your classifier. Rank these variables/ features from most important to the least important. Utilize three strategies covered in the class:

- model reliance
- conditional model reliance
- algorithm reliance

Do all three give the same result?

In [175…

```
# Model Reliance

for i in range(len(C_range)):
    prob = svm_problem(y_train,X_train)
    param = svm_parameter('-s 0 -t 0 -c {}'.format(C_range[i]))
    model = svm_train(prob, param)

    rho=model.rho[0]
    w = np.matmul(X_train[np.array(model.get_sv_indices()) - 1].T, model.get_sv_coef())
    b = -model.rho.contents.value

    p_labels, p_acc, p_vals = svm_predict(y_test, X_test, model)
    for i = 1:len('CulmenLength')
        pi =

print(p_labels)
```

```
Accuracy = 28.9855% (20/69) (classification)
Accuracy = 69.5652% (48/69) (classification)
Accuracy = 92.7536% (64/69) (classification)
Accuracy = 94.2029% (65/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
Accuracy = 95.6522% (66/69) (classification)
[2.0, 2.0, 2.0, 0.0, 2.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 2.
0, 2.0, 1.0, 0.0, 1.0, 1.0, 2.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 2.0,
1.0, 0.0, 2.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 2.0, 1.0, 2.0, 2.0, 0.0, 0.0, 1.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0, 1.0
, 0.0]
```

In [ ]:

Processing math: 100%

Boosting a decision stump

The goal of this notebook is to implement your own boosting module.

- Go through an implementation of decision trees.
- Implement Adaboost ensembling.
- Use your implementation of Adaboost to train a boosted decision stump ensemble.
- Evaluate the effect of boosting (adding more decision stumps) on performance of the model.
- Explore the robustness of Adaboost to overfitting.

This file is adapted from course material by Carlos Guestrin and Emily Fox.

Let's get started!

Import some libraries

In [250…

```
## please make sure that the packages are updated to the newest version.

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Getting the data ready

Load the dataset.

In [251…

```
loans = pd.read_csv('loan_small.csv')
```

Recoding the target column

We re-assign the target to have +1 as a safe (good) loan, and -1 as a risky (bad) loan. In the next cell, the features are also briefly explained.

In [252…

```
features = ['grade',              # grade of the loan
            'term',               # the term of the loan
            'home_ownership',     # home ownership status: own, mortgage or rent
            'emp_length',         # number of years of employment
           ]

loans['safe_loans'] = loans['loan_status'].apply(lambda x : +1 if x=='Fully Paid' else -1)

## please update pandas to the newest version in order to execute the following line
loans.drop(columns=['loan_status'], inplace=True)

target = 'safe_loans' # this variable will be used later

loans.info()
loans.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 40000 entries, 0 to 39999
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   term            40000 non-null  object
 1   grade           40000 non-null  object
 2   home_ownership  40000 non-null  object
 3   emp_length      38167 non-null  object
 4   safe_loans      40000 non-null  int64 
dtypes: int64(1), object(4)
memory usage: 1.5+ MB
```

Out[252…

	term	grade	home_ownership	emp_length	safe_loans
0	36 months	A	RENT	8 years	1
1	60 months	E	MORTGAGE	10+ years	1
2	36 months	D	RENT	10+ years	1
3	36 months	B	RENT	< 1 year	1
4	36 months	F	RENT	7 years	1

Transform categorical data into binary features

In this assignment, we will work with **binary decision trees**. Since all of our features are currently categorical features, we want to turn them into binary features using 1-hot encoding.

We can do so with the following code block:

In [253…

```
## pandas.get_dummies(data, prefix=Nont, prefix_sep'_',dummy_na=False, columns = None, sparse=False, drop_first=F
## Convert categorical variable into dummy/indicator variables
loans = pd.get_dummies(loans)
loans.head()
```

Out[253…

	safe_loans	term_ 36 months	term_ 60 months	grade_A	grade_B	grade_C	grade_D	grade_E	grade_F	grade_G	...	emp_length_10+ years	emp_length_2 years	emp_le
0	1	1	0	1	0	0	0	0	0	0	...	0	0	
1	1	0	1	0	0	0	0	1	0	0	...	1	0	
2	1	1	0	0	0	0	1	0	0	0	...	1	0	
3	1	1	0	0	1	0	0	0	0	0	...	0	0	
4	1	1	0	0	0	0	0	0	1	0	...	0	0	

5 rows × 26 columns

Let's see what the feature columns look like now:

In [254…

```
features = list(loans.columns)
features.remove('safe_loans')  # Remove the response variable
features
```

Out[254…

```
['term_ 36 months',
 'term_ 60 months',
 'grade_A',
 'grade_B',
 'grade_C',
 'grade_D',
 'grade_E',
 'grade_F',
 'grade_G',
 'home_ownership_MORTGAGE',
 'home_ownership_NONE',
 'home_ownership_OTHER',
 'home_ownership_OWN',
 'home_ownership_RENT',
 'emp_length_1 year',
 'emp_length_10+ years',
 'emp_length_2 years',
 'emp_length_3 years',
 'emp_length_4 years',
 'emp_length_5 years',
 'emp_length_6 years',
 'emp_length_7 years',
 'emp_length_8 years',
 'emp_length_9 years',
 'emp_length_< 1 year']
```

Train-test split

We split the data into training and test sets with 80% of the data in the training set and 20% of the data in the test set. We use `seed=1` so that everyone gets the same result.

In [255…

```
# Split arrays or matirces into random train and test subsets
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(loans, test_size = 0.2, random_state=1)
```

Weighted decision trees

Since the data weights change as we build an AdaBoost model, we need to first code a decision tree that supports weighting of individual data points.

Weighted error definition

Consider a model with $N$ data points with:

- Predictions $\hat{y}_1 ... \hat{y}_n$
- Target $y_1 ... y_n$
- Data point weights $\alpha_1 ... \alpha_n$.

Then the **weighted error** is defined by: $$ \mathrm{E}(\mathbf{\alpha}, \mathbf{\hat{y}}) = \frac{\sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y_i}]}{\sum_{i=1}^{n} \alpha_i} $$ where $1[y_i \neq \hat{y_i}]$ is an indicator function that is set to $1$ if $y_i \neq \hat{y_i}$.

Write a function to compute weight of mistakes

Write a function that calculates the weight of mistakes for making the "weighted-majority" predictions for a dataset. The function accepts two inputs:

- `labels_in_node` : Targets $y_1 ... y_n$
- `data_weights` : Data point weights $\alpha_1 ... \alpha_n$

We are interested in computing the (total) weight of mistakes, i.e. $$ \mathrm{WM}(\mathbf{\alpha}, \mathbf{\hat{y}}) = \sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y_i}]. $$ This quantity is analogous to the number of mistakes, except that each mistake now carries different weight. It is related to the weighted error in the following way: $$ \mathrm{E}(\mathbf{\alpha}, \mathbf{\hat{y}}) = \frac{\mathrm{WM}(\mathbf{\alpha}, \mathbf{\hat{y}})}{\sum_{i=1}^{n} \alpha_i} $$

The function **intermediate_node_weighted_mistakes** should first compute two weights:

- $\mathrm{WM}_{-1}$: weight of mistakes when all predictions are $\hat{y}_i = -1$ i.e $\mathrm{WM}(\mathbf{\alpha}, \mathbf{-1}$)
- $\mathrm{WM}_{+1}$: weight of mistakes when all predictions are $\hat{y}_i = +1$ i.e $\mbox{WM}(\mathbf{\alpha}, \mathbf{+1}$)

  where $\mathbf{-1}$ and $\mathbf{+1}$ are vectors where all values are -1 and +1 respectively.

After computing $\mathrm{WM}_{-1}$ and $\mathrm{WM}_{+1}$, the function **intermediate_node_weighted_mistakes** should return the lower of the two weights of mistakes, along with the class associated with that weight. We have provided a skeleton for you with `YOUR CODE HERE` to be filled in several places.

In [256…

```python
def intermediate_node_weighted_mistakes(labels_in_node, data_weights):
    # Sum the weights of all entries with label +1
    total_weight_positive = sum(data_weights[labels_in_node == +1])

    # Weight of mistakes for predicting all -1's is equal to the sum above
    ### YOUR CODE HERE

    WM_n = sum(total_weight_positive *labels_in_node[labels_in_node == +1] )
    ...

    # Sum the weights of all entries with label -1
    ### YOUR CODE HERE
    total_weight_negative = sum(data_weights[labels_in_node == -1])
    ...

    # Weight of mistakes for predicting all +1's is equal to the sum above
    ### YOUR CODE HERE
    ...
    WM_p = sum(total_weight_negative * labels_in_node[labels_in_node == -1])

    # Return the tuple (weight, class_label) representing the lower of the two weights
    #    class_label should be an integer of value +1 or -1.
    # If the two weights are identical, return (weighted_mistakes_all_positive,+1)
    ### YOUR CODE HERE
    if total_weight_positive >= total_weight_negative:
        return (total_weight_negative,+1)
    else:
        return (total_weight_positive,-1)
    ...
```

Checkpoint: Test your **intermediate_node_weighted_mistakes** function, run the following cell:

In [257…

```python
example_labels = pd.Series([-1, -1, 1, 1, 1])
example_data_weights = pd.Series([1., 2., .5, 1., 1.])

if intermediate_node_weighted_mistakes(example_labels, example_data_weights) == (2.5, -1):
    print('Test passed!')
else:
    print('Test failed... try again!')
```

```
Test passed!
```

Recall that the **classification error** is defined as follows: $$ \mbox{classification error} = \frac{\mbox{# mistakes}}{\mbox{# all data points}} $$

Function to pick best feature to split on

The next step is to pick the best feature to split on.

The **best_splitting_feature** function takes the data, the festures, the targetm and the data weights as input and returns the best feature to split on.

Complete the following function.

In [258…

```
# If the data is identical in each feature, this function should return None

def best_splitting_feature(data, features, target, data_weights):

    # These variables will keep track of the best feature and the corresponding error
    best_feature = None
    best_error = float('+inf')
    num_points = float(len(data))

    # Loop through each feature to consider splitting on that feature
    for feature in features:

        # The left split will have all data points where the feature value is 0
        # The right split will have all data points where the feature value is 1
        left_split = data[data[feature] == 0]
        right_split = data[data[feature] == 1]

        # Apply the same filtering to data_weights to create left_data_weights, right_data_weights
        ## YOUR CODE HERE
        left_data_weights = data_weights[data[feature]==0]
        right_data_weights = data_weights[data[feature]==1]
        ...

        # Calculate the weight of mistakes for left and right sides
        ## YOUR CODE HERE
        left_weighted_mistakes, left_class = intermediate_node_weighted_mistakes(np.array(left_split[target]),np.
        right_weighted_mistakes, right_class = intermediate_node_weighted_mistakes(np.array(right_split[target]),
        ...

        # Compute weighted error by computing
        #  ( [weight of mistakes (left)] + [weight of mistakes (right)] ) / [total weight of all data points]
        ## YOUR CODE HERE
        error = (left_weighted_mistakes + right_weighted_mistakes)/sum(data_weights)
        ...

        # If this is the best error we have found so far, store the feature and the error
        if error < best_error:
            best_feature = feature
            best_error = error

    # Return the best feature we found
    return best_feature
```

Checkpoint: Now, we have another checkpoint to make sure you are on the right track.

In [259…

```
example_data_weights = np.array(len(train_data)* [1.5])
if best_splitting_feature(train_data, features, target, example_data_weights) == 'term_ 36 months':
    print('Test passed!')
else:
    print('Test failed... try again!')
```

```
Test passed!
```

Aside. Relationship between weighted error and weight of mistakes:

By definition, the weighted error is the weight of mistakes divided by the weight of all data points, so $$ \mathrm{E}(\mathbf{\alpha}, \mathbf{\hat{y}}) = \frac{\sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y_i}]}{\sum_{i=1}^{n} \alpha_i} = \frac{\mathrm{WM}(\mathbf{\alpha}, \mathbf{\hat{y}})}{\sum_{i=1}^{n} \alpha_i}. $$

In the code above, we obtain $\mathrm{E}(\mathbf{\alpha}, \mathbf{\hat{y}})$ from the two weights of mistakes from both sides, $\mathrm{WM}(\mathbf{\alpha}_{\mathrm{left}}, \mathbf{\hat{y}}_{\mathrm{left}})$ and $\mathrm{WM}(\mathbf{\alpha}_{\mathrm{right}}, \mathbf{\hat{y}}_{\mathrm{right}})$. First, notice that the overall weight of mistakes $\mathrm{WM}(\mathbf{\alpha}, \mathbf{\hat{y}})$ can be broken into two weights of mistakes over either side of the split: $$ \mathrm{WM}(\mathbf{\alpha}, \mathbf{\hat{y}}) = \sum_{i=1}^{n} \alpha_i \times 1[y_i \neq \hat{y_i}] = \sum_{\mathrm{left}} \alpha_i \times 1[y_i \neq \hat{y_i}] + \sum_{\mathrm{right}} \alpha_i \times 1[y_i \neq \hat{y_i}]\\ = \mathrm{WM}(\mathbf{\alpha}_{\mathrm{left}}, \mathbf{\hat{y}}_{\mathrm{left}}) + \mathrm{WM}(\mathbf{\alpha}_{\mathrm{right}}, \mathbf{\hat{y}}_{\mathrm{right}}) $$ We then divide through by the total weight of all data points to obtain $\mathrm{E}({\alpha}, \mathbf{\hat{y}})$: $$ \mathrm{E}({\alpha}, \mathbf{\hat{y}}) = \frac{\mathrm{WM}({\alpha}_{\mathrm{left}}, \mathbf{\hat{y}}_{\mathrm{left}}) + \mathrm{WM}({\alpha}_{\mathrm{right}}, \mathbf{\hat{y}}_{\mathrm{right}})}{\sum_{i=1}^{n} \alpha_i} $$

Building the tree

With the above functions implemented correctly, we are now ready to build our decision tree. A decision tree will be represented as a dictionary which contains the following keys:

```
{
   'is_leaf'            : True/False.
   'prediction'         : Prediction at the leaf node.
   'left'               : (dictionary corresponding to the left tree).
   'right'              : (dictionary corresponding to the right tree).
   'features_remaining' : List of features that are posible splits.
}
```

Let us start with a function that creates a leaf node given a set of target values:

In [260…

```
def create_leaf(target_values, data_weights):

    # Create a leaf node
    leaf = {'splitting_feature' : None,
            'is_leaf': True}

    # Computed weight of mistakes.
    weighted_error, best_class = intermediate_node_weighted_mistakes(target_values, data_weights)
    # Store the predicted class (1 or -1) in leaf['prediction']
    ## YOUR CODE HERE
    leaf['prediction'] = best_class
    ...

    return leaf
```

We provide a function that learns a weighted decision tree recursively and implements 3 stopping conditions:

1. All data points in a node are from the same class.
2. No more features to split on.
3. Stop growing the tree when the tree depth reaches **max_depth**.

In [261…

```
def weighted_decision_tree_create(data, features, target, data_weights, current_depth = 1, max_depth = 10):
    remaining_features = features[:] # Make a copy of the features.
    target_values = data[target]
    #data['data_weights']=data_weights
    print("--------------------------------------------------------------------")
    print("Subtree, depth = %s (%s data points)." % (current_depth, len(target_values)))

    # Stopping condition 1. Error is 0.
    if intermediate_node_weighted_mistakes(target_values, data_weights)[0] <= 1e-15:
        print("Stopping condition 1 reached.")
        return create_leaf(target_values, data_weights)

    # Stopping condition 2. No more features.
    if remaining_features == []:
        print("Stopping condition 2 reached.")
        return create_leaf(target_values, data_weights)

    # Additional stopping condition (limit tree depth)
    if current_depth > max_depth:
        print("Reached maximum depth. Stopping for now.")
        return create_leaf(target_values, data_weights)

    # If all the datapoints are the same, splitting_feature will be None. Create a leaf
    splitting_feature = best_splitting_feature(data, features, target, data_weights)
    remaining_features.remove(splitting_feature)

    left_split = data[data[splitting_feature] == 0]
    right_split = data[data[splitting_feature] == 1]

    left_data_weights = data_weights[data[splitting_feature] == 0]
    right_data_weights = data_weights[data[splitting_feature] == 1]

    print("Split on feature %s. (%s, %s)" % (\
              splitting_feature, len(left_split), len(right_split)))

    # Create a leaf node if the split is "perfect"
    if len(left_split) == len(data):
        print("Creating leaf node.")
        return create_leaf(left_split[target], data_weights)
    if len(right_split) == len(data):
        print("Creating leaf node.")
        return create_leaf(right_split[target], data_weights)

    # Repeat (recurse) on left and right subtrees
    ## YOUR CODE HERE
    left_tree = weighted_decision_tree_create(
        left_split, remaining_features, target, left_data_weights, current_depth+1, max_depth)
    right_tree = weighted_decision_tree_create(
        right_split, remaining_features, target, right_data_weights, current_depth+1, max_depth)


    return {'is_leaf'          : False,
            'prediction'       : None,
            'splitting_feature': splitting_feature,
            'left'             : left_tree,
            'right'            : right_tree}
```

Here is a recursive function to count the nodes in your tree:

In [262…

```
def count_nodes(tree):
    if tree['is_leaf']:
        return 1
    return 1 + count_nodes(tree['left']) + count_nodes(tree['right'])
```

Run the following test code to check your implementation. Make sure you get **'Test passed'** before proceeding.

In [263…

```
example_data_weights = np.array([1.0 for i in range(len(train_data))])
small_data_decision_tree = weighted_decision_tree_create(train_data, features, target,
                                        example_data_weights, max_depth=2)
if count_nodes(small_data_decision_tree) == 7:
    print('Test passed!')
else:
    print('Test failed... try again!')
    print('Number of nodes found:', count_nodes(small_data_decision_tree))
    print('Number of nodes that should be there: 7')
```

```
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Split on feature grade_A. (8775, 75)
--------------------------------------------------------------------
Subtree, depth = 3 (8775 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (75 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Split on feature grade_D. (19331, 3819)
--------------------------------------------------------------------
Subtree, depth = 3 (19331 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (3819 data points).
Reached maximum depth. Stopping for now.
Test passed!
```

Let us take a quick look at what the trained tree is like. You should get something that looks like the following

```
{'is_leaf': False,
    'left': {'is_leaf': False,
        'left': {'is_leaf': True, 'prediction': -1, 'splitting_feature': None},
        'prediction': None,
        'right': {'is_leaf': True, 'prediction': 1, 'splitting_feature': None},
        'splitting_feature': 'grade_A'
     },
    'prediction': None,
    'right': {'is_leaf': False,
        'left': {'is_leaf': True, 'prediction': 1, 'splitting_feature': None},
        'prediction': None,
        'right': {'is_leaf': True, 'prediction': -1, 'splitting_feature': None},
        'splitting_feature': 'grade_D'
     },
     'splitting_feature': 'term. 36 months'
}
```

In [264…

```
small_data_decision_tree
```

Out[264…

```
{'is_leaf': False,
 'prediction': None,
 'splitting_feature': 'term_ 36 months',
 'left': {'is_leaf': False,
  'prediction': None,
  'splitting_feature': 'grade_A',
  'left': {'splitting_feature': None, 'is_leaf': True, 'prediction': -1},
  'right': {'splitting_feature': None, 'is_leaf': True, 'prediction': 1}},
 'right': {'is_leaf': False,
  'prediction': None,
  'splitting_feature': 'grade_D',
  'left': {'splitting_feature': None, 'is_leaf': True, 'prediction': 1},
  'right': {'splitting_feature': None, 'is_leaf': True, 'prediction': -1}}}
```

Making predictions with a weighted decision tree

We give you a function that classifies one data point. It can also return the probability if you want to play around with that as well.

In [265…

```
def classify(tree, x, annotate = False):
    # If the node is a leaf node.
    if tree['is_leaf']:
        if annotate:
            print("At leaf, predicting %s" % tree['prediction'])
        return tree['prediction']
    else:
        # Split on feature.
        split_feature_value = x[tree['splitting_feature']]
        if annotate:
            print("Split on %s = %s" % (tree['splitting_feature'], split_feature_value))
        if split_feature_value == 0:
            return classify(tree['left'], x, annotate)
        else:
            return classify(tree['right'], x, annotate)
```

Evaluating the tree

Now, we will write a function to evaluate a decision tree by computing the classification error of the tree on the given dataset.

Again, recall that the **classification error** is defined as follows: $$ \mbox{classification error} = \frac{\mbox{# mistakes}}{\mbox{# all data points}} $$

The function called **evaluate_classification_error** takes in as input:

1. `tree` (as described above)
2. `data` (a dataframe)

The function does not change because of adding data point weights.

In [266…

```
def evaluate_classification_error(tree, data):
    # Apply the classify(tree, x) to each row in your data
    # YOUR CODE HERE
    prediction = data.apply(lambda x: classify(tree,x), axis =1)
    ...

    # Once you've made the predictions, calculate the classification error
    return (prediction != data[target]).sum() / float(len(data))
```

In [267…

```
evaluate_classification_error(small_data_decision_tree, test_data)
```

Out[267…

```
0.390875
```

Example: Training a weighted decision tree

To build intuition on how weighted data points affect the tree being built, consider the following:

Suppose we only care about making good predictions for the **first 10 and last 10 items** in `train_data`, we assign weights:

- 1 to the last 10 items
- 1 to the first 10 items
- and 0 to the rest.

Let us fit a weighted decision tree with `max_depth = 2`.

In [268…

```
# Assign weights
example_data_weights = np.array([1.] * 10 + [0.]*(len(train_data) - 20) + [1.] * 10)

# Train a weighted decision tree model.
small_data_decision_tree_subset_20 = weighted_decision_tree_create(train_data, features, target,
                         example_data_weights, max_depth=2)
```

```
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature emp_length_10+ years. (22413, 9587)
--------------------------------------------------------------------
Subtree, depth = 2 (22413 data points).
Split on feature grade_A. (19673, 2740)
--------------------------------------------------------------------
Subtree, depth = 3 (19673 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 3 (2740 data points).
Stopping condition 1 reached.
--------------------------------------------------------------------
Subtree, depth = 2 (9587 data points).
Stopping condition 1 reached.
```

Now, we will compute the classification error on the `subset_20`, i.e. the subset of data points whose weight is 1 (namely the first and last 10 data points).

In [269…

```
subset_20 = train_data.head(10).append(train_data.tail(10))
evaluate_classification_error(small_data_decision_tree_subset_20, subset_20)
```

Out[269…

```
0.15
```

Now, let us compare the classification error of the model `small_data_decision_tree_subset_20` on the entire test set `train_data`:

In [270…

```
evaluate_classification_error(small_data_decision_tree_subset_20, train_data)
```

Out[270…

```
0.445625
```

The model `small_data_decision_tree_subset_20` performs **a lot** better on `subset_20` than on `train_data`.

So, what does this mean?

- The points with higher weights are the ones that are more important during the training process of the weighted decision tree.
- The points with zero weights are basically ignored during training.

Implementing your own Adaboost (on decision stumps)

Now that we have a weighted decision tree working, it takes only a bit of work to implement Adaboost. For the sake of simplicity, let us stick with **decision tree stumps** by training trees with **`max_depth=1`**.

Recall from the lecture notes the procedure for Adaboost:

1. Start with unweighted data with $\alpha_j = 1$

2. For t = 1,...T:

- Learn $f_t(x)$ with data weights $\alpha_j$
- Compute coefficient $\hat{w}_t$: $$\hat{w}_t = \frac{1}{2}\ln{\left(\frac{1- \mbox{E}(\mathbf{\alpha}, \mathbf{\hat{y}})}{\mbox{E}

(\mathbf{\alpha}, \mathbf{\hat{y}})}\right)}$$

- Re-compute weights $\alpha_j$: $$\alpha_j \gets \begin{cases} \alpha_j \exp{(-\hat{w}_t)} & \text{ if }f_t(x_j) = y_j\\ \alpha_j \exp{(\hat{w}_t)} & \text{ if }f_t(x_j) \neq y_j \end{cases}$$
- Normalize weights $\alpha_j$: $$\alpha_j \gets \frac{\alpha_j}{\sum_{i=1}^{N}{\alpha_i}} $$

Complete the skeleton for the following code to implement **adaboost_with_tree_stumps**. Fill in the places with `YOUR CODE HERE`.

In [271…

```
from math import log
from math import exp

def adaboost_with_tree_stumps(data, features, target, num_tree_stumps):
    # start with unweighted data (uniformly weighted)
    alpha =  np.array([1.]*len(data))
    weights = []
    tree_stumps = []
    target_values = data[target]

    for t in range(num_tree_stumps):
        print('=====================================================')
        print('Adaboost Iteration %d' % t)
        print('=====================================================')
        # Learn a weighted decision tree stump. Use max_depth=1
        # YOUR CODE HERE
        tree_stump = weighted_decision_tree_create(data, features, target, data_weights = alpha, max_depth =1 )
        tree_stumps.append(tree_stump)
        ...

        # Make predictions
        ## YOUR CODE HERE
        predictions = data.apply(lambda x: classify(tree_stump, x), axis =1)
        ...

        # Produce a Boolean array indicating whether
        # each data point was correctly classified
        is_correct = predictions == target_values
        is_wrong   = predictions != target_values

        # Compute weighted error
        ## YOUR CODE HERE
        weighted_error = np.sum(np.array(is_wrong)*alpha)/np.sum(alpha)
        ...

        # Compute model coefficient using weighted error
        ## YOUR CODE HERE
        weight = (1/2)*log((1-weighted_error)/(weighted_error))
        weights.append(weight)
        ...

        # Adjust weights on data point
        ## YOUR CODE HERE
        adjustment = is_correct.apply(lambda is_correct : exp(-weight) if is_correct else exp(weight))

        # Scale alpha by multiplying by adjustment
        # Then normalize data points weights
        ## YOUR CODE HERE
        alpha = alpha*adjustment
        alpha = alpha/sum(alpha)
        ...

    return weights, tree_stumps
```

Checking your Adaboost code

Train an ensemble of **two** tree stumps and see which features those stumps split on. We will run the algorithm with the following parameters:

- `train_data`
- `features`
- `target`
- `num_tree_stumps = 2`

In [272…

```
stump_weights, tree_stumps = adaboost_with_tree_stumps(train_data, features, target, num_tree_stumps=2)
```

```
=====================================================
Adaboost Iteration 0
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Reached maximum depth. Stopping for now.
```

```
-------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Reached maximum depth. Stopping for now.
====================================================
Adaboost Iteration 1
====================================================
-------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
-------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
-------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
```

In [273…

```
def print_stump(tree):
    split_name = tree['splitting_feature'] # split_name is something like 'term. 36 months'
    if split_name is None:
        print("(leaf, label: %s)" % tree['prediction'])
        return None
    split_feature, split_value = split_name.rsplit('_',1)
    print('                       root')
    print('         |---------------|----------------|')
    print('         |                                |')
    print('         |                                |')
    print('         |                                |')
    print('  [{0} == 0]{1}[{0} == 1]    '.format(split_name, ' '*(27-len(split_name))))
    print('         |                                |')
    print('         |                                |')
    print('         |                                |')
    print('    (%s)                 (%s)' \
        % (('leaf, label: ' + str(tree['left']['prediction']) if tree['left']['is_leaf'] else 'subtree'),
           ('leaf, label: ' + str(tree['right']['prediction']) if tree['right']['is_leaf'] else 'subtree')))
```

Here is what the first stump looks like:

In [274…

```
print_stump(tree_stumps[0])
```

```
                       root
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [term_ 36 months == 0]            [term_ 36 months == 1]    
         |                                |
         |                                |
         |                                |
    (leaf, label: -1)                 (leaf, label: 1)
```

Here is what the next stump looks like:

In [275…

```
print_stump(tree_stumps[1])
```

```
                       root
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [grade_A == 0]                    [grade_A == 1]    
         |                                |
         |                                |
         |                                |
    (leaf, label: -1)                 (leaf, label: 1)
```

In [276…

```
print(stump_weights)
```

```
[0.17198848113764034, 0.17728780637267785]
```

If your Adaboost is correctly implemented, the following things should be true:

- `tree_stumps[0]` should split on **term. 36 months** with the prediction -1 on the left and +1 on the right.
- `tree_stumps[1]` should split on **grade.A** with the prediction -1 on the left and +1 on the right.

- Weights should be approximately `[0.17, 0.18]`

Reminders

- Stump weights ($\mathbf{\hat{w}}$) and data point weights ($\mathbf{\alpha}$) are two different concepts.
- Stump weights ($\mathbf{\hat{w}}$) tell you how important each stump is while making predictions with the entire boosted ensemble.
- Data point weights ($\mathbf{\alpha}$) tell you how important each data point is while training a decision stump.

Training a boosted ensemble of 10 stumps

Let us train an ensemble of 10 decision tree stumps with Adaboost. We run the **adaboost_with_tree_stumps** function with the following parameters:

- `train_data`
- `features`
- `target`
- `num_tree_stumps = 10`

In [277…

```
stump_weights, tree_stumps = adaboost_with_tree_stumps(train_data, features,
                                target, num_tree_stumps=10)
```

```
=====================================================
Adaboost Iteration 0
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 1
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 2
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_D. (26027, 5973)
--------------------------------------------------------------------
Subtree, depth = 2 (26027 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (5973 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 3
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_B. (23457, 8543)
--------------------------------------------------------------------
Subtree, depth = 2 (23457 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (8543 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 4
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_E. (28766, 3234)
--------------------------------------------------------------------
Subtree, depth = 2 (28766 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3234 data points).
Reached maximum depth. Stopping for now.
```

```
=====================================================
Adaboost Iteration 5
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature home_ownership_MORTGAGE. (16870, 15130)
--------------------------------------------------------------------
Subtree, depth = 2 (16870 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (15130 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 6
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 7
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 8
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 9
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_E. (28766, 3234)
--------------------------------------------------------------------
Subtree, depth = 2 (28766 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3234 data points).
Reached maximum depth. Stopping for now.
```

Plot the boosted stumps in the additive model

The decision stumps picks a feature and a threshold, visualize them here.

In [244…

```
for stump in tree_stumps:
    print_stump(stump)
```

```
                       root
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [term_ 36 months == 0]            [term_ 36 months == 1]
         |                                |
         |                                |
         |                                |
    (leaf, label: -1)                 (leaf, label: 1)
                       root
         |---------------|----------------|
         |                                |
         |                                |
         |                                |
  [grade_A == 0]                    [grade_A == 1]
```

```
       |                                |
       |                                |
       |                                |
  (leaf, label: -1)                 (leaf, label: 1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_D == 0]                    [grade_D == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: 1)                 (leaf, label: -1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_B == 0]                    [grade_B == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: -1)                 (leaf, label: 1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_E == 0]                    [grade_E == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: 1)                 (leaf, label: -1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[home_ownership_MORTGAGE == 0]    [home_ownership_MORTGAGE == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: -1)                 (leaf, label: 1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_A == 0]                    [grade_A == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: -1)                 (leaf, label: 1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_F == 0]                    [grade_F == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: 1)                 (leaf, label: -1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_A == 0]                    [grade_A == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: -1)                 (leaf, label: 1)
                     root
       |---------------|----------------|
       |                                |
       |                                |
       |                                |
[grade_E == 0]                    [grade_E == 1]
       |                                |
       |                                |
       |                                |
  (leaf, label: 1)                 (leaf, label: -1)
```

Making predictions

Making predictions

Recall from the lecture that in order to make predictions, we use the following formula: $$ \hat{y} = sign\left(\sum_{t=1}^T \hat{w}_t f_t(x)\right) $$

We need to do the following things:

- Compute the predictions $f_t(x)$ using the $t$-th decision tree
- Compute $\hat{w}_t f_t(x)$ by multiplying the `stump_weights` with the predictions $f_t(x)$ from the decision trees
- Sum the weighted predictions over each stump in the ensemble.

Complete the following skeleton for making predictions:

In [278…

```
def predict_adaboost(stump_weights, tree_stumps, data):
    scores = np.array([0.]*len(data))

    for i, tree_stump in enumerate(tree_stumps):
        predictions = data.apply(lambda x: classify(tree_stump, x), axis = 1)

        # Accumulate predictions on scaores array
        # YOUR CODE HERE
        scores = scores + (predictions * stump_weights[i])
        ...

    return scores.apply(lambda score : +1 if score > 0 else -1)
```

In [279…

```
predictions = predict_adaboost(stump_weights, tree_stumps, test_data)

from sklearn.metrics import accuracy_score
accuracy = accuracy_score(test_data[target], predictions)
print('Accuracy of 10-component ensemble = %s' % accuracy)
```

```
Accuracy of 10-component ensemble = 0.62825
```

Now, let us take a quick look what the `stump_weights` look like at the end of each iteration of the 10-stump ensemble:

In [280…

```
stump_weights
```

Out[280…

```
[0.17198848113764034,
 0.17728780637267785,
 0.10308067696993935,
 0.08686702058341694,
 0.07220085937787622,
 0.07438562925255676,
 0.05834552873231902,
 0.045454870264690556,
 0.03194548460003601,
 0.023305292432214228]
```

Question i: Are the weights monotonically decreasing, monotonically increasing, or neither?

The weights are Monotonically decreasing.

Reminder: Stump weights ($\mathbf{\hat{w}}$) tell you how important each stump is while making predictions with the entire boosted ensemble.

Performance plots

In this section, we will try to reproduce some performance plots.

How does accuracy change with adding stumps to the ensemble?

We will now train an ensemble with:

- `train_data`
- `features`
- `target`
- `num_tree_stumps = 30`

Once we are done with this, we will then do the following:

- Compute the classification error at the end of each iteration.
- Plot a curve of classification error vs iteration.

First, lets train the model.

In [281…

```
# this may take a while...
stump_weights, tree_stumps = adaboost_with_tree_stumps(train_data,
                                 features, target, num_tree_stumps=30)
```

```
=====================================================
Adaboost Iteration 0
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 1
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 2
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_D. (26027, 5973)
--------------------------------------------------------------------
Subtree, depth = 2 (26027 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (5973 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 3
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_B. (23457, 8543)
--------------------------------------------------------------------
Subtree, depth = 2 (23457 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (8543 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 4
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_E. (28766, 3234)
--------------------------------------------------------------------
Subtree, depth = 2 (28766 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3234 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 5
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature home_ownership_MORTGAGE. (16870, 15130)
--------------------------------------------------------------------
Subtree, depth = 2 (16870 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (15130 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 6
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
```

```
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 7
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 8
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 9
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_E. (28766, 3234)
--------------------------------------------------------------------
Subtree, depth = 2 (28766 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3234 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 10
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 11
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 12
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature emp_length_10+ years. (22413, 9587)
--------------------------------------------------------------------
Subtree, depth = 2 (22413 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (9587 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 13
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_B. (23457, 8543)
--------------------------------------------------------------------
Subtree, depth = 2 (23457 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
```

```
Subtree, depth = 2 (8543 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 14
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 15
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_D. (26027, 5973)
--------------------------------------------------------------------
Subtree, depth = 2 (26027 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (5973 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 16
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 17
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 18
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_E. (28766, 3234)
--------------------------------------------------------------------
Subtree, depth = 2 (28766 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3234 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 19
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_C. (23388, 8612)
--------------------------------------------------------------------
Subtree, depth = 2 (23388 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (8612 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 20
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature home_ownership_MORTGAGE. (16870, 15130)
--------------------------------------------------------------------
Subtree, depth = 2 (16870 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (15130 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 21
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature term_ 36 months. (8850, 23150)
--------------------------------------------------------------------
Subtree, depth = 2 (8850 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (23150 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 22
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_F. (30624, 1376)
--------------------------------------------------------------------
Subtree, depth = 2 (30624 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (1376 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 23
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_B. (23457, 8543)
--------------------------------------------------------------------
Subtree, depth = 2 (23457 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (8543 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 24
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature emp_length_2 years. (29104, 2896)
--------------------------------------------------------------------
Subtree, depth = 2 (29104 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (2896 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 25
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_G. (31657, 343)
--------------------------------------------------------------------
Subtree, depth = 2 (31657 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (343 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 26
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_A. (28081, 3919)
--------------------------------------------------------------------
Subtree, depth = 2 (28081 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (3919 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 27
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_G. (31657, 343)
--------------------------------------------------------------------
Subtree, depth = 2 (31657 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (343 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 28
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature home_ownership_OWN. (29204, 2796)
--------------------------------------------------------------------
Subtree, depth = 2 (29204 data points).
```

```
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (2796 data points).
Reached maximum depth. Stopping for now.
=====================================================
Adaboost Iteration 29
=====================================================
--------------------------------------------------------------------
Subtree, depth = 1 (32000 data points).
Split on feature grade_G. (31657, 343)
--------------------------------------------------------------------
Subtree, depth = 2 (31657 data points).
Reached maximum depth. Stopping for now.
--------------------------------------------------------------------
Subtree, depth = 2 (343 data points).
Reached maximum depth. Stopping for now.
```

Computing training error at the end of each iteration

Now, we will compute the classification error on the **train_data** and see how it is reduced as trees are added.

In [282…

```
error_all = []
for n in range(1, 31):
    predictions = predict_adaboost(stump_weights[:n], tree_stumps[:n], train_data)
    error = 1.0 - accuracy_score(train_data[target], predictions)
    error_all.append(error)
    print("Iteration %s, training error = %s" % (n, error_all[n-1]))
```

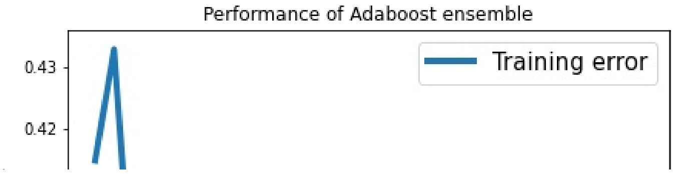
```
Iteration 1, training error = 0.41484374999999996
Iteration 2, training error = 0.43281250000000004
Iteration 3, training error = 0.39059374999999996
Iteration 4, training error = 0.39059374999999996
Iteration 5, training error = 0.37931250000000005
Iteration 6, training error = 0.38228125
Iteration 7, training error = 0.37253125
Iteration 8, training error = 0.37549999999999994
Iteration 9, training error = 0.37253125
Iteration 10, training error = 0.37253125
Iteration 11, training error = 0.37253125
Iteration 12, training error = 0.37150000000000005
Iteration 13, training error = 0.37253125
Iteration 14, training error = 0.37150000000000005
Iteration 15, training error = 0.37150000000000005
Iteration 16, training error = 0.37150000000000005
Iteration 17, training error = 0.37150000000000005
Iteration 18, training error = 0.37146875
Iteration 19, training error = 0.37150000000000005
Iteration 20, training error = 0.37146875
Iteration 21, training error = 0.37209375
Iteration 22, training error = 0.37146875
Iteration 23, training error = 0.37212500000000004
Iteration 24, training error = 0.37150000000000005
Iteration 25, training error = 0.37150000000000005
Iteration 26, training error = 0.37212500000000004
Iteration 27, training error = 0.37150000000000005
Iteration 28, training error = 0.37131250000000005
Iteration 29, training error = 0.37121875000000004
Iteration 30, training error = 0.37124999999999997
```

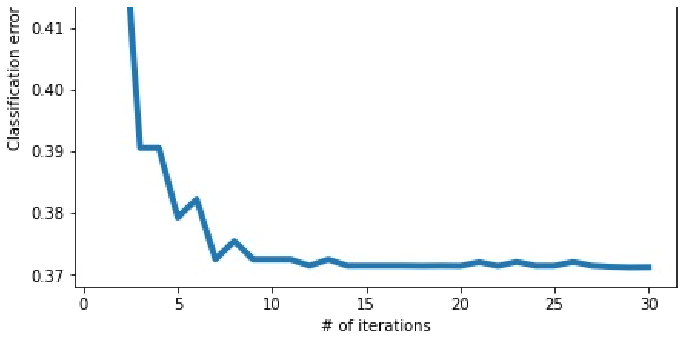
Visualizing training error vs number of iterations

We have provided you with a simple code snippet that plots classification error with the number of iterations.

In [283…

```
plt.rcParams['figure.figsize'] = 7, 5
plt.plot(list(range(1,31)), error_all, '-', linewidth=4.0, label='Training error')
plt.title('Performance of Adaboost ensemble')
plt.xlabel('# of iterations')
plt.ylabel('Classification error')
plt.legend(loc='best', prop={'size':15})

plt.rcParams.update({'font.size': 16})
```





Evaluation on the test data

Performing well on the training data is cheating, so lets make sure it works on the `test_data` as well. Here, we will compute the classification error on the `test_data` at the end of each iteration.

In [284…

```
test_error_all = []
for n in range(1, 31):
    predictions = predict_adaboost(stump_weights[:n], tree_stumps[:n], test_data)
    error = 1.0 - accuracy_score(test_data[target], predictions)
    test_error_all.append(error)
    print("Iteration %s, test error = %s" % (n, test_error_all[n-1]))
```

```
Iteration 1, test error = 0.41037500000000005
Iteration 2, test error = 0.43174999999999997
Iteration 3, test error = 0.390875
Iteration 4, test error = 0.390875
Iteration 5, test error = 0.37825
Iteration 6, test error = 0.382625
Iteration 7, test error = 0.37175
Iteration 8, test error = 0.37612500000000004
Iteration 9, test error = 0.37175
Iteration 10, test error = 0.37175
Iteration 11, test error = 0.37175
Iteration 12, test error = 0.369375
Iteration 13, test error = 0.369375
Iteration 14, test error = 0.369375
Iteration 15, test error = 0.369375
Iteration 16, test error = 0.369375
Iteration 17, test error = 0.369375
Iteration 18, test error = 0.371
Iteration 19, test error = 0.369375
Iteration 20, test error = 0.371
Iteration 21, test error = 0.36924999999999997
Iteration 22, test error = 0.371
Iteration 23, test error = 0.367625
Iteration 24, test error = 0.369375
Iteration 25, test error = 0.369375
Iteration 26, test error = 0.367625
Iteration 27, test error = 0.369375
Iteration 28, test error = 0.36950000000000005
Iteration 29, test error = 0.369
Iteration 30, test error = 0.369
```

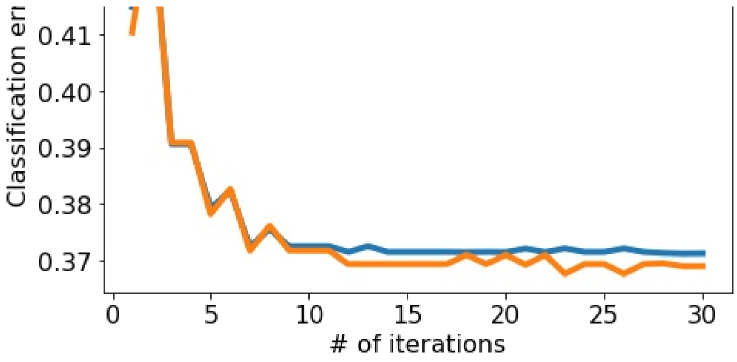
Visualize both the training and test errors

Now, let us plot the training & test error with the number of iterations.

In [285…

```
plt.rcParams['figure.figsize'] = 7, 5
plt.plot(list(range(1,31)), error_all, '-', linewidth=4.0, label='Training error')
plt.plot(list(range(1,31)), test_error_all, '-', linewidth=4.0, label='Test error')

plt.title('Performance of Adaboost ensemble')
plt.xlabel('# of iterations')
plt.ylabel('Classification error')
plt.rcParams.update({'font.size': 16})
plt.legend(loc='best', prop={'size':15})
plt.tight_layout()
```

Performance of Adaboost ensemble
0.43
0.42
ror
Training error
Test error



Question ii: From this plot (with 30 trees), is there massive overfitting as the # of iterations increases?

In [ ]:

```
No,it is not.
```