APRENDIZADO DE MÁQUINA - TRABALHO 03

Sistemas de Recomendação e Aprendizado de Comitês

Jefferson Colares

Análise e conclusões sobre o código Python disponibilizado para este trabalho.

Professor:
Eduardo Bezerra

Rio de Janeiro,
Agosto, 2018

Sumário

Parte 1 - Introdução

Esse trabalho aborda os temas Filtragem Colaborativa e Aprendizado de Comitês.

Mantive essa breve introdução para esse relatório apresentar uma estrutura consistente com a do enunciado do trabalho. Desta forma, por exemplo, o item 2.3 desse relatório corresponde ao item 2.3 do enunciado.

1.1 Localização dos arquivos

O código referente a parte de sistemas de recomendação pode ser encontrado nos arquivos **parte1/main.py** e **parte1/cofi_cost_func.py**.

O código referente a aprendizado de comitês está no arquivo **parte2/ensembles_sklearn.py**.

Parte 2 - Sistemas de Recomendação

Implementação do sistema de recomendação de filmes utilizando filtragem colaborativa.

2.1 Conjunto de dados de classificações de filme

Para a construção desse sistema, utilizaremos um conjunto de dados com as seguintes características e objetos:

Número de usuários ($n_{(u)}$): 943

Número de filmes ($n_{(m)}$): 1682

Número de características: 100

No arquivo ex8_movies.mat:

Matriz $Y_{(1682x943)}$, contendo as classificações (de 1 a 5) dadas pelos usuários a cada filme:

$$\begin{bmatrix} Y^{(1,1)} & ... & Y^{(1,943)} \\ ... & ... & ... \\ Y^{(1682,1)} & ... & Y^{(1682,943)} \end{bmatrix}$$

Matriz $R_{(1682x943)}$, contendo apenas 0s e 1s, sendo 1 quando um usuário houver classificado um filme ou 0 em caso contrário. Posteriormente, para facilitar a computação, os elementos dessa matriz são convertidos em False e True.

$$\begin{bmatrix} R^{(1,1)} & ... & R^{(1,943)} \\ ... & ... & ... \\ R^{(1682,1)} & ... & R^{(1682,943)} \end{bmatrix}$$

No arquivo ex8_movieParams.mat:

Matrix $X_{(1682x100)}$, contendo em cada linha o vetor de características de cada filme.

$$\begin{bmatrix} X^{(1,1)} & ... & X^{(1,100)} \\ ... & ... & ... \\ X^{(1682,1)} & ... & X^{(1682,100)} \end{bmatrix}$$

Matrix $\Theta_{(1682x100)}$, contendo em cada linha o vetor de parâmetros correspondente a cada usuário.

$$\begin{bmatrix} \Theta^{(1,1)} & ... & \Theta^{(1,100)} \\ ... & ... & ... \\ \Theta^{(943,1)} & ... & \Theta^{(943,100)} \end{bmatrix}$$

Arquivo movie_ids.txt: Lista com os nomes de todos os filmes do conjunto de dados e seus respectivos números.

2.2 Algoritmo de aprendizagem de filtragem colaborativa

A predição da avaliação dos filmes pelos usuários é dada através do produto $X\Theta^T$. Antes, porém, é preciso aprender os valores de X e Θ através de filtragem colaborativa. Para esse aprendizado, precisaremos das funções de custo e gradiente, cujas implementações são analisadas a seguir.

2.2.1 Função de custo da filtragem colaborativa

$$J(x^{(i)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2$$

A expressão acima define a função de custo. Como já aprendemos, dei preferencia a uma implementação vetorizada, para maior eficiência computacional. O código ficou assim:

```
#Calculo do custo (J)
cost_sq = np.power((np.dot(X, Theta.T) - Y), 2)
cost_sq_rated = np.multiply(cost_sq,  R)
J = np.sum(cost_sq_rated) / 2
```

A matriz cost_sq recebe o cálculo do custo (squared error). Na linha seguinte, descartamos do cálculo os filmes que não receberam avaliações.

O resultado (22.224604) bate com o resultado esperado de acordo com o enunciado.

Posteriormente verifiquei que o gradiente não convergia corretamente. Por esse motivo, inclui o termo de regularização no cálculo do custo. O código adicionado foi:

```
#Parcela de regularizacao do custo
reg =  Lambda/2 * (np.sum(np.power(Theta, 2)) + np.sum(np.power(X, 2)))
#Custo regularizado
J = J + reg
```

2.2.2 Gradiente de filtragem colaborativa

Na filtragem colaborativa, estimamos θ em função de x e x em função de θ, minimizando ambos simultaneamente. Os gradientes da função de custo são calculados pelas expressões abaixo.

$$\frac{\partial J}{\partial x_k^i} = \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})\theta_k^{(j)}$$

$$\frac{\partial J}{\partial \theta_k^i} = \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(j)}$$

De forma vetorizada, essas expressões podem ser representadas respectivamente por :

$$((X\Theta^T - Y)^T \odot R)^T\Theta \quad \text{e} \quad ((X\Theta^T - Y)^T \odot R)^T X$$

Obs.: Nas expressões acima, utilizei a notação $\odot$ para representar uma multiplicação elemento-a-elemento (element-wise) entre duas matrizes com as mesmas dimensões. No caso, essa multiplicação ponto-a-ponto foi utilizada para o cálculo contemplar apenas os filmes marcados como avaliados (True) por cada usuário na matriz R.

A implementação das duas expressões em Python ficou assim:

```
#Calculo dos gradientes
cost = np.dot(X, Theta.T) - Y
cost_rated = np.multiply(cost, R)
Theta_grad = np.dot(cost_rated.T, X)
X_grad = np.dot(cost_rated, Theta)
```

Posteriormente, para viabilizar a convergência da função de minimização, precisei acrescentar os termos de regularização, conforme abaixo:

```
# Calculo dos gradientes com regularizacao
Theta_grad = np.dot(cost_rated.T, X) + np.dot(Theta, Lambda)
X_grad = np.dot(cost_rated, Theta) + np.dot(X, Lambda)
```

A saída do programa após o treinamento foi a seguinte:

```
Treinamento da filtragem colaborativa...
Optimization terminated successfully.
         Current function value: 35118.790099
         Iterations: 305
         Function evaluations: 460
         Gradient evaluations: 460
Aprendizado do Sistema de Recomendacao finalizado
```

2.3 Aprendizado de Recomendações para Filmes

Depois de preparar a função de custo e gradiente (os dois cálculos precisam ser fornecidos pela mesma função, que é passada como parâmetro para a função de otimização), podemos ajustar os parâmetros e fazer o treinamento do modelo.

Utilizando a filtragem colaborativa, as features dos filmes são estimadas de acordo com as preferências dos usuários e vice-versa, simultaneamente, durante a fase de aprendizado.

As atividades de treinamento e o código correspondente são analisados a seguir:

1. Carga dos dados. As matrizes Y e R (contendo as avaliações dos usuários) são carregadas a partir do arquivo ex8_movies.mat.

```
#  Carga dos dados
data = scipy.io.loadmat('../data/ex8_movies.mat')
Y = data['Y']
R = data['R'].astype(bool)
```

   Nesse ponto, é possível inserir avaliações de filmes produzidas por um usuário em uma coluna adicional nas matrizes Y e R para depois obter recomendações personalizadas. O trecho de código abaixo insere essa nova coluna na posição 0:

```
#  Inicia o vetor de avaliacoes do novo usuario
my_ratings = np.zeros(1682)
my_ratings[0] = 4
my_ratings[97] = 2
my_ratings[6] = 3
my_ratings[11] = 5
my_ratings[53] = 4
my_ratings[63] = 5
my_ratings[65] = 3
my_ratings[68] = 5
my_ratings[182] = 4
my_ratings[225] = 5
my_ratings[354] = 5
#  Adiciona algumas avaliacoes a matriz
Y = np.column_stack((my_ratings, Y))
R = np.column_stack((my_ratings, R)).astype(bool)
```

2. Normalização das avaliações. A normalização é uma etapa de pré-processamento importante para o funcionamento eficiente de algoritimos de minimização como o que vamos utilizar. Entretanto, nesse caso, tem uma função a mais: Podemos utilizar a média obtida para poder obter recomendações "default"para usuários que não avaliaram nenhum filme.

```
# Normaliza avaliacoes
Ynorm, Ymean = normalize_ratings(Y, R)
```

3. Inicialização de variáveis. Algumas variáveis que serão utilizadas posteriormente são inicializadas aqui.

   As matrizes X e Theta são inicializadas com valores aleatórios para garantir que o algoritmo aprenda features diferentes umas das outras. Depois são reformatadas como arrays unidimensionais e armazenadas na array initial_parameters.

   O hiperparâmetro lambda, que determina o fator de regularização também é inicializado nesta parte do código.

```
num_users = Y.shape[1]
num_movies = Y.shape[0]
num_features = 10
# Define parametros iniciais (Theta, X)
X = np.random.rand(num_movies, num_features)
Theta = np.random.rand(num_users, num_features)
initial_parameters = np.hstack((X.T.flatten(), Theta.T.flatten()))
# fator de regularizacao
Lambda = 10
```

4. Funções de custo e gradiente. Nesse trecho do código, o operador *lambda* do Python é utilizado para armazenar as chamadas nossa função cofi_cost_func() duas vezes, com os nomes costFunc e gradFunc. Essas chamadas serão utilizadas como parâmetros na etapa seguinte.

```
costFunc = lambda p: cofi_cost_func(p, Ynorm, R, num_users, num_movies,
           num_features, Lambda)[0]
gradFunc = lambda p: cofi_cost_func(p, Ynorm, R, num_users, num_movies,
           num_features, Lambda)[1]
```

5. Função de minimização. A linha seguinte do código chama a função scipy.optimize() que executa o gradiente descendente. O resultado da otimização é armazenado na variável *result*, de onde em seguida são extraídas as matrizes X e Theta.

```
result = minimize(costFunc, initial_parameters, method='CG', jac=gradFunc, options={'disp': T
theta = result.x
cost = result.fun
# Extrai as matrizes X e Theta a partir de theta
X = theta[:num_movies*num_features].reshape(num_movies, num_features)
Theta = theta[num_movies*num_features:].reshape(num_users, num_features)
```

Com o modelo treinado, é possível realizar recomendações. A recomendação é feita multiplicando Θ por X e selecionando os filmes que atingiram a maior pontuação.

```
p = X.dot(Theta.T)
my_predictions = p[:, 0] + Ymean
```

Observe no trecho de código acima que:

- Cada coluna da matriz **p** contém as predições para um usuário. Nesse exemplo, selecionamos apenas as previsões para o usuário da coluna 0.
- Os valores em **p** estão normalizados e precisam ser somados com média (Ymean) para obtermos as notas das recomendações.

  Um usuário que não fez nenhuma avaliação de filmes teria sua coluna totalmente zerada. Com a adição da média, podemos fornecer a ele uma recomendação baseada na média das preferências dos demais usuários.

Por fim, o trecho de código a seguir classifica as recomendações para o usuário 0 (armazenadas anteriormente no vetor *my_predictions*) em ordem decrescente, imprime os nomes dos filmes correspondentes às 10 primeiras linhas e imprime as avaliações fornecidas anteriormente pelo usuário 0.

A função load_movie_list() chamada na primeira linha do código lê o arquivo movie_ids.txt e armazena o resultado em uma matriz contendo o id e a descrição dos filmes em cada uma de suas duas colunas.

```
movieList = load_movie_list()

# ordena predicoes em ordem decrescente
pre=np.array([[idx, p] for idx, p in enumerate(my_predictions)])
post = pre[pre[:,1].argsort()[::-1]]
r = post[:,1]
ix = post[:,0]

print('\nRecomendacoes principais:')
for i in range(10):
    j = int(ix[i])
    print('\tPrevisao de avaliacao %.1f para %s' % (my_predictions[j], movieList[j]))

print('\nAvaliacoes originais fornecidas:')
for i in range(len(my_ratings)):
    if my_ratings[i] > 0:
        print('\tAvaliou %d para %s' % (my_ratings[i], movieList[i]))
```

O resultado é impresso na tela como no exemplo abaixo:

```
Recomendacoes principais:
        Previsao de avaliacao 5.0 para Great Day in Harlem, A (1994)
        Previsao de avaliacao 4.6 para To Kill a Mockingbird (1962)
        Previsao de avaliacao 4.6 para Singin in the Rain (1952)
        Previsao de avaliacao 4.6 para Barcelona (1994)
        Previsao de avaliacao 4.6 para Amadeus (1984)
        Previsao de avaliacao 4.5 para Princess Bride, The (1987)
        Previsao de avaliacao 4.5 para Maya Lin: A Strong Clear Vision (1994)
        Previsao de avaliacao 4.5 para Schindlers List (1993)
        Previsao de avaliacao 4.5 para Casablanca (1942)
        Previsao de avaliacao 4.4 para Wrong Trousers, The (1993)

Avaliacoes originais fornecidas:
        Avaliou 4 para Toy Story (1995)
        Avaliou 3 para Twelve Monkeys (1995)
        Avaliou 5 para Usual Suspects, The (1995)
        Avaliou 4 para Outbreak (1995)
        Avaliou 5 para Shawshank Redemption, The (1994)
        Avaliou 3 para While You Were Sleeping (1995)
        Avaliou 5 para Forrest Gump (1994)
        Avaliou 2 para Silence of the Lambs, The (1991)
        Avaliou 4 para Alien (1979)
        Avaliou 5 para Die Hard 2 (1990)
        Avaliou 5 para Sphere (1998)
```

Parte 3 - Aprendizado de Comitês

O objetivo, nessa parte do trabalho, é replicar o experimento do Jupyter notebook fornecido junto com o enunciado, porém utilizando o scikit-learn.

O primeiro passo foi analisar o código fornecido:

Inicialmente, é feita a carga dos dados que serão utilizados. O dataset utilizado é o Hastie, que contém 12000 linhas e 11 colunas com 10 features e os labels correspondentes.

```
# Read data
x, y = make_hastie_10_2()
df = pd.DataFrame(x)
df['Y'] = y
```

Em seguida, 20% do dataset é reservado para testes e o restante para treinamento. Os exemplos e os rótulos de cada conjunto são armazenados nas variáveis X_train, Y_train, X_test e Y_test.

```
# Split into training and test set
train, test = train_test_split(df, test_size = 0.2)
X_train, Y_train = train.iloc[:,:-1], train.iloc[:,-1]
X_test, Y_test = test.iloc[:,:-1], test.iloc[:,-1]
```

Na sequência, a função generic_clf() é chamada para criar uma arvore de decisão classificadora. O resultado retornado pela função (as taxas de erro obtidas no teste e no treinamento) são armazenados na variável er_tree.

```
# Fit a simple decision tree first
clf_tree = DecisionTreeClassifier(max_depth = 1, random_state = 1)
er_tree = generic_clf(Y_train, X_train, Y_test, X_test, clf_tree)
```

A função generic_clf() faz o ajuste do modelo da árvore de decisão e faz as predições utilizando o conjunto de treinamento e depois o de teste. Ao final, ela chama a função get_error_rate() que compara as predições com os rótulos (Y) do dataset e calcula as taxas de erro, que são retornadas como resultado da função generic_clf().

```
def generic_clf(Y_train, X_train, Y_test, X_test, clf):
    clf.fit(X_train,Y_train)
    pred_train = clf.predict(X_train)
    pred_test = clf.predict(X_test)
```

```
return get_error_rate(pred_train, Y_train), get_error_rate(pred_test, Y_test)
```

De volta ao fluxo principal, o script agora executa um loop que executa a função adaboost_clf() por 40 vezes. A cada ciclo de execução, a quantidade de interações do adaboost aumenta um pouco (de 10 em 10). Os resultados obtidos, ou seja as taxas de erros obtidas nos conjuntos de treinamento e teste, são acumuladas nas variáveis er_train e er_test, para posterior plotagem no gráfico.

```
# Fit Adaboost classifier using a decision tree as base estimator
# Test with different number of iterations
er_train, er_test = [er_tree[0]], [er_tree[1]]
x_range = range(10, 410, 10)
for i in x_range:
    er_i = adaboost_clf(Y_train, X_train, Y_test, X_test, i, clf_tree)
    er_train.append(er_i[0])
    er_test.append(er_i[1])
```

A função adaboost_clf() é onde ocorre a implementação manual do algoritmo Adaboost. Ela recebe como parâmetros os conjuntos de treinamento e teste, e os objetos M, que é uma variável que irá definir a quantidade de iterações que deverão ser realizadas e clf, que é uma árvore de decisão classificadora (o weak learner a ser potencializado pelo adaboost). As etapas de seu funcionamento são detalhadas a seguir:

Primeiro, um vetor de pesos com o mesmo tamanho do conjunto de treinamento é inicializado com o mesmo valor em todos os seus elementos.

```
w = np.ones(n_train) / n_train
```

Em seguida, inicia o loop que é executado M vezes, onde são realizadas as seguintes computações:

1. Criação e treino de uma árvore de decisão utilizando os pesos no vetor w. Iniciamente, os pesos são iguais para todos os exemplos do conjunto de treimnamento. As predições obtidas por essa árvore nos conjuntos de treinamento e teste são armazenadas nos vetores pred_train_i e pred_test_i

```
# Fit a classifier with the specific weights
clf.fit(X_train, Y_train, sample_weight = w)
pred_train_i = clf.predict(X_train)
pred_test_i = clf.predict(X_test)
```

2. Os vetores *miss* e *miss2* registram os erros de predição do modelo. No *miss* são registrados 1 para erros e 0 para acertos. No *miss2*, os acertos são convertidos em -1.

```
# Indicator function
```

```
miss = [int(x) for x in (pred_train_i != Y_train)]
# Equivalent with 1/-1 to update weights
miss2 = [x if x==1 else -1 for x in miss]
```

3. A variável err_m recebe a média ponderada dos erros do modelo.

```
# Error
err_m = np.dot(w,miss) / sum(w)
```

4. Com base na taxa de erros, define o valor da taxa de atualização do modelo, alpha.

```
# Alpha
alpha_m = 0.5 * np.log( (1 - err_m) / float(err_m))
```

5. Atualiza os pesos, de modo que as predições erradas tenham um peso maior que os acertos.

```
# New weights
w = np.multiply(w, np.exp([float(x) * alpha_m for x in miss2]))
```

Aumentar os pesos dos exemplos classificados de forma errada faz com que o método DecisionTreeClassifier.fit() dê a eles maior atenção a eles cada vez que é executado pelo adaboost. A estratégia de aumentar progressivamente a quantidade de vezes que a árvore é ajustada a cada execução do adaboost também colabora para melhorar o ajuste do modelo e diminuir sua taxa de erro.

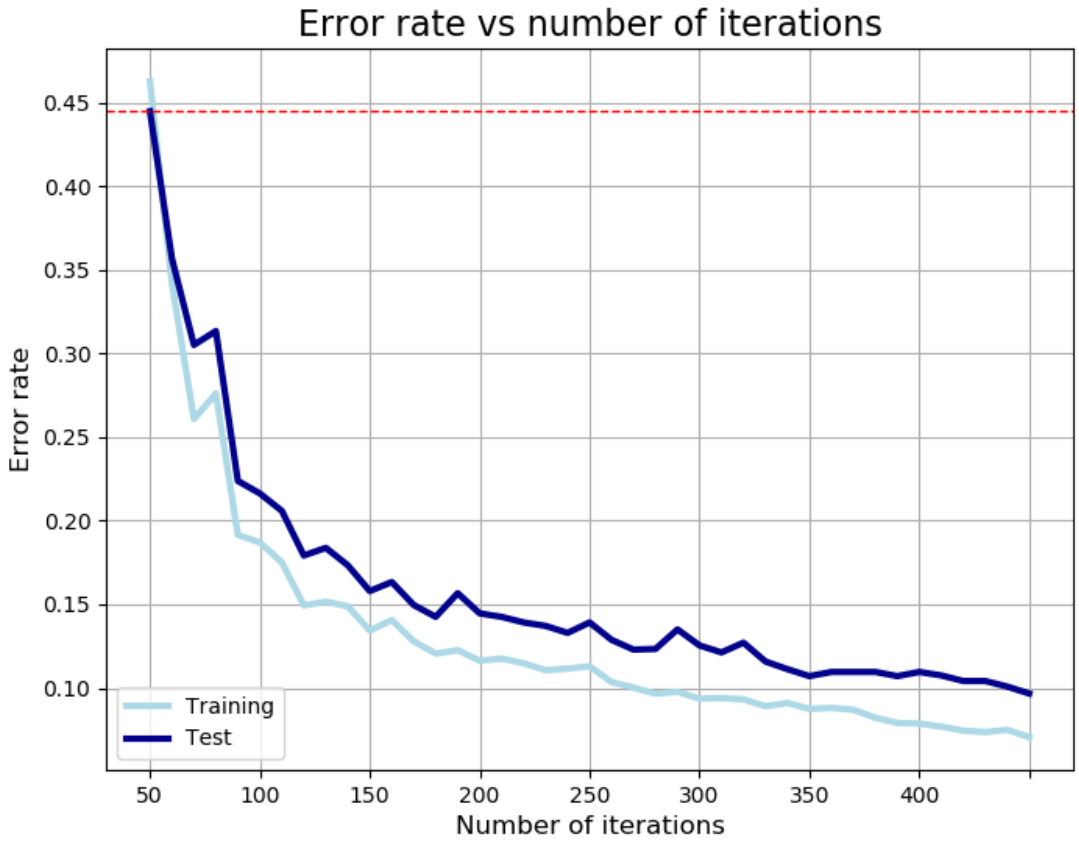
O gráfico gerado pelo programa mostra como a repetição adaptativa do adaboost diminui progressivamente a taxa inicial de erro da árvore de decisão.

Depois de entender o funcionamento do programa fornecido, fiz algumas adaptações, para executar o mesmo experimento utilizando o adaboost da classe **sklearn.ensemble.AdaBoostClassifier**.

Mantive todo o código original, para possibilitar a geração do gráfico comparando a curva de aprendizado dos dois métodos. Nos parágrafos abaixo, comento apenas o código que eu adicionei.

Primeiro eu criei uma nova função, chamada adaboost_ada(). De modo semelhante à função original, ela recebe como parâmetros os conjuntos de teste e treinamento, e ainda uma variável M que define a quantidade de arvores de decisão que deverão ser instanciadas a cada execução do algoritmo.

O AdaBoostClassifier() utiliza como default o DecisionTreeClassifier como weak learner, então, diferente do programa original, não foi necessário me preocupar com instanciação e atribuição de pesos aos exemplos.



A função incializa M arvores de decisão, com uma taxa de aprendizado fixa = 1, executa o treinamento e armazena retorna a taxa média de erro obtida nos conjuntos de treinamento e teste, conforme pode ser verificado no código abaixo:

```
def adaboost_ada(Y_train, X_train, Y_test, X_test, M):
    ada = AdaBoostClassifier(n_estimators=M, learning_rate=1)
    ada = ada.fit(X_train, Y_train)
    score_train = ada.score(X_train, Y_train)
    score_test = ada.score(X_test, Y_test)
    # Return error rate in train and test set
    return(1 - score_train, 1 - score_test)
```

Além dessa função, criei uma nova versão da função de plotagem do gráfico, preparada agora para imprimir quatro linhas: as duas originais e mais duas (verdes) com o desempenho do aprendizado da nova função de classificação. Reproduzo abaixo apenas as duas linhas modificadas.

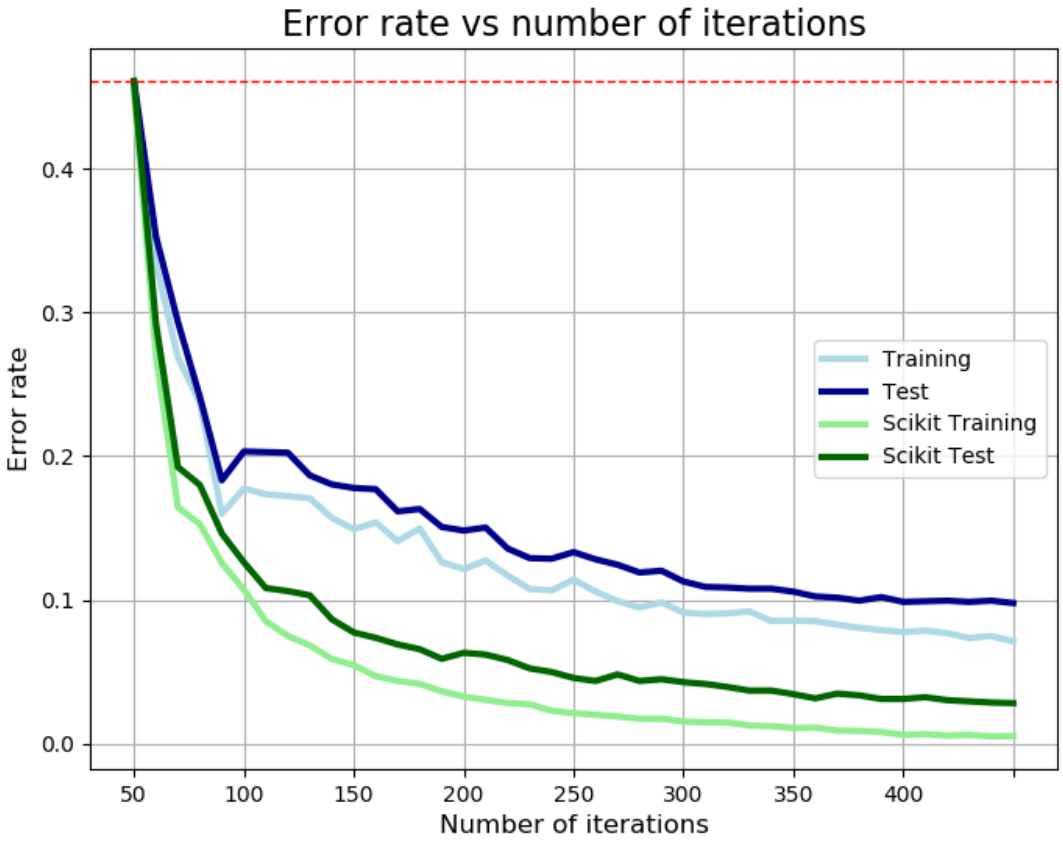
```
df_error.columns = ['Training', 'Test', 'Scikit Training', 'Scikit Test']
    plot1 = df_error.plot(linewidth = 3, figsize = (8,6), color = ['lightblue', 'darkblue','lightg
```

Por fim, na função main(), implementei um pequeno trecho de código que executa a função do adabost repetidas vezes, aumentando progressivamente a quantidade de árvores utilizadas.

As taxas de erro de treinamento e teste são coletadas a cada novo ciclo para posterior plotagem no gráfico. Segue abaixo o código implementado:

```
# Adaboost ada - utilizando AdaBoostClassifier
er_train_ada, er_test_ada = [er_tree[0]], [er_tree[1]]
new_range = range(10, 401, 10)
for i in new_range:
    er_i = adaboost_ada(Y_train, X_train, Y_test, X_test, i)
    er_train_ada.append(er_i[0])
    er_test_ada.append(er_i[1])
```

Como pode ser visto abaixo, no novo gráfico gerado pelo programa, pode-se observar que substituir a implementação "manual"do adaboost pela fornecida no Scikit-Learn, proporciona um resultado (acurácia) melhor e uma convergência mais rápida. Além disso, o tempo de execução é muito menor.



Na última atividade desse trabalho, implementei o adaboost utilizando bagging. Diferente das abordagens anteriores, no bagging, várias amostras diferentes obtidas a partir do conjunto de dados de treinamento são utilizadas para treinar o weak learner.

A implementação foi semelhante à anterior, de modo a possibilitar a comparação entre as três abordagens de utilização de comitês. Apenas a classe utilizada mudou, pois nesse caso foi

utilizada a sklearn.ensemble.BaggingClassifier.

Com exceção de pequenas diferenças em nomes de variáveis, a única mudança feita em relação ao modelo inicial foi na linha que aciona o classificador, que reproduzo abaixo junto com o restante da nova função:

```
def adaboost_bag(Y_train, X_train, Y_test, X_test, M):
    bag = BaggingClassifier(n_estimators=M)
    bag = bag.fit(X_train, Y_train)
    score_train = bag.score(X_train, Y_train)
    score_test = bag.score(X_test, Y_test)
    # Return error rate in train and test set
    return(1 - score_train, 1 - score_test)
```

Pude observar que o bagging, ao menos utilizando as configurações default, consome mais recursos do computador que os demais, demorando mais a completar a execução.

Finalizando, o gráfico a seguir demonstra que o o algoritmo de bagging não apresenta ganhos de performance quando executado com maior quantidade de decision trees.