Trabalho Prático 2 - Implemetação do Algoritmo de Boosting

Disciplina: Aprendizado de Máquina

Aluno: Danilo Pimentel de Carvalho Costa

Matrícula: 2016058077

Introdução

Duas tentativas de implementar o algoritmo de boosting foram feitas. Como será mostrado a seguir, nenhuma das duas implementações teve grande successo em termos de acurácia. Os resultados serão discutidos nas seções a seguir.

As duas implementações utilizam as ferramentas que a biblioteca sklearn fornece para construção de classificadores. Tais ferramentas foram convenientes para utilizar as funções de avaliação de modelos que a biblioteca também fornece. A referência consultada se encontra nesta página: [https://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator (https://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator)](https://scikit-learn.org/stable/developers/develop.html#rolling-your-own-estimator)

In [33]:

```
# Dependencies

import math
import numpy as np
import pandas as pd

from sklearn.preprocessing import label_binarize
from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import cross_val_score

from sklearn.base import BaseEstimator, ClassifierMixin

from sklearn.utils.validation import check_X_y, check_array, check_is_fitted

# Method used to evaluate performance of the classifiers.
# Reference: https://scikit-learn.org/stable/modules/cross_validation.html#compu
ting-cross-validated-metrics
def eval_classifier_performance(clf, X, y):
    scores = cross_val_score(clf, X, y, cv=5)
    print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

Tentativa 1 - Abordagem ingênua

A primeira tentativa de implementação adota uma estratégia ingênua, utilizando estruturas de repetição, condições manuais, e lambdas em seu funcionamento. Tal implementação se provou simples de se construir, porém muito lenta em termos de performance. A hipótese é de que todas as estruturas de repetição, checagens e chamadas de função lambda adicionam um fardo na execução. A má performance mostrada na subseção "Avaliação e Performance" tornou inviável sua depuração e avaliação.

Implementação

In [34]:

```
class Tp2ClassifierTake1(BaseEstimator, ClassifierMixin):
    def __init__(self, n_stumps=10):
        self.n_stumps = n_stumps

    def fit(self, X, y):
        # Check that X and y have correct shape
        X, y = check_X_y(X, y, dtype='str')

        self.X_ = pd.DataFrame(X)
        self.y_ = y

        # Create stumps based on the categories found for each feature...
        stumps = []
        ## ...iterating over the features of the dataset...
        for (name, column) in self.X_.iteritems():
            # ...getting the unique categories for each feature...
            values = column.unique()
            for value in values:
                # ...and finally creating lambda functions to return true when
                # the example has the given category for the given feature...
                stumps.append(lambda c: 1 if c[name] == value else -1)
                # ...and the same for the ausence of the given category for the
                # given feature.
                stumps.append(lambda c: 1 if c[name] != value else -1)
        # Also create lambdas for the stumps that return fixed values.
        stumps.append(lambda _: 1)
        stumps.append(lambda _: -1)

        # The number of given examples.
        m = len(y)

        # We then create the weights array, with the same weigth for all the
        # examples. This array is going to be updated according to the mistakes
        # made the selected stump on each iteration.
        w = [1 / m] * m
        # The array below will store tupls of the stumps (lambdas) that we
        # select during and the respective importance.
        selected_stumps = []
        for i in range(self.n_stumps):
            # We look for the best stump by calculating the weighted empirical
            # error.
            best_stump = None
            best_stump_empirical_error = -1
            for stump in stumps:
                empirical_error = 0
                for k, item in self.X_.iterrows():
                    if stump(item) != y[k]:
                        empirical_error += w[k]
                if best_stump == None or best_stump_empirical_error > empirical_
error:
                    best_stump = stump
                    best_stump_empirical_error = empirical_error

            # After finding the best stump for the current weights, we calculate
its
            # importance.
            if best_stump_empirical_error == 0:
                # If it happens, we found a strong classifier for these example
s. We
```

```
                # consider its importance as 1, and stop looking for more stump
s.
                selected_stumps.append((1, best_stump))
                break
            else:
                # If the stump made mistakes, we calculate the importance regula
rly.
                alpha = (1 / 2) * math.log((1 - best_stump_empirical_error) / be
st_stump_empirical_error)
                selected_stumps.append((alpha, best_stump))

            # Knowing the selected stump mistakes, we can now update the weights
so we
            # choose a stump that is good on what the selected stump is bad.
            w_total = 0
            for j, item in self.X_.iterrows():
                w[j] *= math.exp(-1 * alpha * best_stump(item) * y[j])
                w_total += w[j]
            for j in range(m):
                w[j] /= w_total

        # We store the selected stumps for futher prediction.
        self.selected_stumps_ = selected_stumps

        # Return the classifier
        return self

    def predict(self, X):
        # Check is fit had been called.
        check_is_fitted(self)

        # Input validation.
        X = check_array(X, ensure_2d=False, dtype='str')

        # Make predictions for each example contained on X.
        y_predicted = []
        for i, item in pd.DataFrame(X).iterrows():
            # Calculate the score based on the selected stump responses
            # considering its importances.
            score = 0
            for selected_stump in self.selected_stumps_:
                alpha, stump = selected_stump
                score += alpha * stump(item)

            # If the final score is negative, the predict negative. Predict
            # positive otherwise.
            prediction = -1 if score < 0 else 1
            y_predicted.append(prediction)

        return y_predicted

# Create some simple data for testing the implementation
dummy_X = np.array([
    ['x', 'o'],
    ['o', 'o'],
    ['o', 'x']
])
dummy_y = np.array([
    1,
    -1,
    1
```

```
])
dummy_predict_X = np.array([
    ['x', 'x'],
    ['o', 'o']
])

Tp2ClassifierTake1(n_stumps=1).fit(dummy_X, dummy_y).predict(dummy_predict_X)
```

Out[34]:

```
[1, -1]
```

Avaliação e Performance

Não foi preciso uma análise mais elaborada para entender que o classificador implementado acima não tem boa performance. Mesmo configurando para selecionar somente 1 stump, a execução da validação cruzada leva mais de 10 segundos. Aumentando para 5 stumps, o tempo de execução salta para mais de 1 minuto. Os resultados obtidos nas validações cruzadas executadas não são conclusivos, mas a evidência de má performance é suficiente para motivar a busca por uma melhor implementação.

In [35]:

```
# Load the tic tac toe dataset
tic_tac_toe_data = pd.read_csv('./tic-tac-toe/tic-tac-toe.data', header=None)

# Create the matrix X with the examples
X = tic_tac_toe_data.drop([9], axis=1)

# Transform the classes from strings to -1s and 1s
y = tic_tac_toe_data[9]
y = label_binarize(y, classes=['negative', 'positive'], pos_label = 1, neg_label
= -1)
y = y.ravel()
```

Validação Cruzada 5-fold selecionando 1 stump

In [36]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake1(n_stumps=1), X, y)
```

```
Accuracy: 0.65 (+/- 0.00)
CPU times: user 14.8 s, sys: 22.3 ms, total: 14.8 s
Wall time: 14.9 s
```

Validação Cruzada 5-fold selecionando 5 stumps

In [37]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake1(n_stumps=5), X, y)
```

```
Accuracy: 0.65 (+/- 0.00)
CPU times: user 1min 13s, sys: 106 ms, total: 1min 13s
Wall time: 1min 13s
```

Tentativa 2 - Abordagem utilizando matrizes

Para melhorar o tempo de treinamento e predição do classificador, a implementação foi repensada para utilizar a boa performance de operações matriciais utilizando a biblioteca numpy. Os ganhos de performance foram expressivos, o que possibilitou a melhor depuração, evolução, e avaliação do algoritmo.

Implementação

In [38]:

```
class Tp2ClassifierTake2(BaseEstimator, ClassifierMixin):
    def __init__(self, n_stumps=10):
        self.n_stumps = n_stumps

    # X is the dataset. S is the matrix with stumps for each feature. See the fi
t
    # method for the description of how to build S.
    def compute_stump_predictions_(self, X, S):
        # Compute all stumps predictions for all examples given. We will try to
 build
        # a matrix P that will contain 0s indicating that the stump returned fal
se for
        # example, and 1 otherwise.
        P = np.matmul(X, np.transpose(S)) # dimensions: m x [amount of stumps]
        # Here we select the values that indicate stumps returning true
        P = np.ma.masked_array(
            P,
            # We are removing from P...
            np.logical_and(
                # ...category-presence stumps returning false...
                P != 1,
                # ...and category-ausence stumps return false.
                np.logical_not(np.logical_and(P < 0, P != -1))
            )
        )
        # We fill the values we removed with 0s to indicate that the stumps retu
rned
        # false...
        P = np.ma.filled(P, fill_value=0)
        # ...everything else with 1s to indicate that the stump returned true.
        P[P != 0] = 1

        # The part of setting the predictions for stumps that return always true
or
        # always false is left for the method caller, as it changes between the
 fit
        # and the predict methods.
        return P

    # We are assuming here that X will be a matrix of positive integers (no zero
s,
    # no negatives). A good enhancement to be made here is to accept any kind of
class,
    # and then process it to fit the expectations mentioned. y needs to be a bin
ary
    # vector (0s and 1s).
    def fit(self, X, y):
        # Check that X and y have correct shape.
        X, y = check_X_y(X, y, dtype='numeric')

        # Number of examples and features.
        m, n = X.shape

        # Number of unique categories.
        c = len(np.unique(X))

        # The amount of stumps: there are two stumps for each category, and we n
eed the
        # stumps for each feature. We also need two stumps with fixed return val
```

```
ue.
        # Therefore, 2 * number of categories * number of features + 2 fixed stu
mps.
        S = np.zeros((2 * c * n + 2, n))

        # Create the stumps that respond true when the example has a given categ
ory for
        # a given feature.
        for j in range(n):
            for k in range(c):
                # The trick here is that we will multiply this value with the ex
ample
                # vector, so if on the given vector, the feature j has the categ
ory
                # k + 1, then the result of the multiplication will be 1. Otherw
ise,
                # it will be something else we can ignore.
                S[j * c + k, j] = 1 / (k + 1)

        # Create the stumps that respond true when the example does not have a g
iven
        # category for a given feature.
        for j in range(n):
            for k in range(c):
                # Following the same thought process, if on the given vector, th
e
                # feature j has the category k + 1, then the result of the
                # multiplication will be -1. Otherwise, it will be something els
e
                # we can USE. We will use any negative result different from -1
 to
                # know that the category k + 1 isn't in there.
                S[(j * c + k) + c * n, j] = -1 / (k + 1)

        # Compute all stumps predictions for all examples given. Check the comme
nts
        # on compute_stump_predictions_ to understand how we do it.
        P = self.compute_stump_predictions_(X, S)
        # We set the output for the two stumps that always return true and
        # false. These are the last 2 stumps on the S matrix.
        P[:, -2] = 0
        P[:, -1] = 1

        # Now we want to know the mistakes made by the stumps. We build a binary
        # matrix E with 0s indicating a correct prediction and 1s indicating err
ors.
        E = (P != y[:, None]) + 0 # dimensions: m x [amount of stumps]

        # As this implementation works with reweighting, we will want a vector w
to
        # store the weights of each example. It starts with equal weight for al
l.
        w = np.array([1 / m] * m)

        # As we select stumps, we need to store its importance and the stump its
elf.
        # The vector A will store the former, while H and Hf will store the latt
er.
        # Hf will indicate if the stump returns a fixed response (1 or -1), or i
ts
        # just a regular stump (0).
```

```
        A = []
        H = []
        Hf = []
        for i in range(self.n_stumps):
            # Compute the stumps weigthed errors.
            EP = np.matmul(w, E) # dimensions: 1 x [amount of stumps]

            # Select the best stump and store it on H
            s = np.argmin(EP)
            H.append(S[s])
            if s == 2 * c * n:
                # Indicate if the stump always return false...
                Hf.append(-1)
            elif s == 2 * c * n + 1:
                # ..., always return true...
                Hf.append(1)
            else:
                # ...or if it is a regular stump.
                Hf.append(0)

            # Compute the selected stump importance.
            et = np.amin(EP)
            if et == 0:
                # If the stump error was 0, it means it is strong for this datas
et.
                # If so, we set its importance to one, and stop selecting stumps
here.
                # This case is always going to happen on the first iteration, as
we
                # always show the full dataset to each stump.
                A.append(1)
                break
            # If the stump made mistakes, then we calculate the importance in th
e
            # regular way and add it to A.
            alpha = (1 / 2) * math.log((1 - et) / et)
            A.append(alpha)

            # Knowing the mistakes that the selected stump made, we need to upda
te the
            # weights so we select a stump that is good on what this one is bad
 at.
            w *= np.vectorize(math.exp)(-1 * alpha * ((((E[:, s] == y) + 0) * 2)
- 1))
            w /= np.sum(w)

        # Now that we selected the stumps, we store them for when we need to pre
dict.
        self.A_ = A
        self.H_ = H
        self.Hf_ = Hf

        # Return the classifier.
        return self

    # The same assumption is made in here. X will be a matrix of positive intege
rs.
    def predict(self, X):
        # Check is fit had been called.
        check_is_fitted(self)
```

```
        # Input validation.
        X = check_array(X, ensure_2d=False, dtype='numeric')

        # We follow the same process to get P with 0s where the stumps returned
 false,
        # and 1s where the stumps returned true.
        P = self.compute_stump_predictions_(X, np.array(self.H_))

        # But now we want it slightly different to make it easier to compute the
final
        # prediction. We keep 1s as they are, but we transform 0s to -1s.
        P = ((P * 2) - 1)

        # We also need to set fixed responses for stumps that always return true
or
        # always return false. We do so with the Hf array we built during traini
ng.
        Hf_np = np.array(self.Hf_)
        m, _ = X.shape
        P[:, Hf_np != 0] = np.tile(Hf_np[Hf_np != 0], (m, 1))

        # To compute the predictions, we sum each stump prediction considering t
he
        # importances we calculated during training. If the result is negative,
 then
        # we predict 0. If the result is positive, then we predict 1.
        y = np.matmul(P, np.array(self.A_))
        y[y >= 0] = 1
        y[y < 0] = 0

        # Finally return the prediction.
        return y

# Create some simple data for testing the implementation
dummy_X = np.array([
    [1, 2],
    [2, 2],
    [2, 1]
])
dummy_y = np.array([
    1,
    0,
    1
])
dummy_predict_X = np.array([
    [1, 1],
    [2, 2]
])

# Test the implementation.
Tp2ClassifierTake2(n_stumps=1).fit(dummy_X, dummy_y).predict(dummy_predict_X)
```

Out[38]:

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

Avaliação e Performance

Na segunda tentativa, podemos observar tempos extremamente menores graças ao uso de operações matriciais. Como mencionado, agora é possível avaliar o algoritmo variando o parâmetro de quantidade de stumps selecionados. Os valores utilizados foram de 1, 5, 10, 50, 100, 500, e 1000 stumps.

Os dados resultantes de tal experimento infelizmente são inconclusivos. A quantidade de stumps selecionados não teve efeito na acurácia do modelo gerado. Claramente há um erro de implementação, e a hipótese é de que ele está no cálculo das importâncias dos stumps e da atualização dos pesos para cada exemplo. Tal erro não foi identificado a tempo da entrega do trabalho.

In [39]:

```
# Load the tic tac toe dataset
tic_tac_toe_data = pd.read_csv('./tic-tac-toe/tic-tac-toe.data', header=None)

# Transform the features categories from strings to positive integers
X = tic_tac_toe_data.drop([9], axis=1)
X = X.apply(LabelEncoder().fit_transform) + 1

# Transform the classes from strings to 0s and 1s
y = tic_tac_toe_data[9]
y = label_binarize(y, classes=['negative', 'positive'], pos_label = 1, neg_label
= 0)
y = y.ravel()
```

In [40]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=1), X, y)
```

```
Accuracy: 0.70 (+/- 0.10)
CPU times: user 142 ms, sys: 5.84 ms, total: 148 ms
Wall time: 45.4 ms
```

In [41]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=5), X, y)
```

```
Accuracy: 0.70 (+/- 0.10)
CPU times: user 157 ms, sys: 5.23 ms, total: 162 ms
Wall time: 45.9 ms
```

In [42]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=10), X, y)
```

```
Accuracy: 0.70 (+/- 0.10)
CPU times: user 174 ms, sys: 5 ms, total: 179 ms
Wall time: 49.6 ms
```

In [43]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=50), X, y)
```

```
Accuracy: 0.70 (+/- 0.10)
CPU times: user 365 ms, sys: 9.22 ms, total: 374 ms
Wall time: 104 ms
```

In [44]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=100), X, y)
```

```
Accuracy: 0.65 (+/- 0.00)
CPU times: user 525 ms, sys: 8.99 ms, total: 534 ms
Wall time: 140 ms
```

In [45]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=500), X, y)
```

```
Accuracy: 0.65 (+/- 0.00)
CPU times: user 1.97 s, sys: 14.6 ms, total: 1.99 s
Wall time: 505 ms
```

In [46]:

```
%%time

eval_classifier_performance(Tp2ClassifierTake2(n_stumps=1000), X, y)
```

```
Accuracy: 0.65 (+/- 0.00)
CPU times: user 3.68 s, sys: 23.4 ms, total: 3.71 s
Wall time: 933 ms
```

Conclusão

A implementação de algoritmos de aprendizagem de máquina deve considerar a grande quantidade de operações realizadas durante o treinamento. Uma implementação que envolve o uso excessivo de estruturas de repetição e controle se provou ineficiente e inutilizável. A adoção de operações matriciais fez grande diferença, uma vez que a execução do treinamento se tornou rápida e viabilizou a realização de experimentos para avaliar a eficácia dos modelos gerados.

Infelizmente não foi possível corrigir a segunda implementação, e por isso não foi possível fazer avaliações conclusivas sobre a eficácia dos modelos criados. Porém, pode-se concluir que a estragégia de Boosting para aprendizado é poderosa, uma vez que pode-se ver acurácias de até 70% conseguidas através de classificadores tão fracos quanto stumps.