report

August 21, 2018

0.1 Matheus Fernandes, Pedro Faustini

0.2 1. Introdução

Este trabalho estuda técnicas de mineração de dados para classificar áudio. Em específico, o universo a ser classificado consiste em dez caracteres: 6, 7, a, b, c, d, h, m, n, x. Cada arquivo de áudio, chamado de *captcha*, contém 4 caracteres, podendo haver repetição. A classificação de um *captcha* é considerada sucesso somente se **todos** os seus quatro caracteres são corretamente identificados.

Para este trabalho foram utilizadas as ferramentas mencionadas abaixo.

- FFMPEG
- Anaconda (Python>=3.5)
- numpy>=1.13.3
- pandas>=0.20.3
- scikit-learn>=0.19.1
- scipy==1.1.0
- librosa>=0.6.1
- matplotlib>=1.5.3
- ipython>=6.2.1
- Sox (`conda install -c conda-forge sox`)

Numpy e Pandas manipulam vetores de uma forma mais otimizada e rica em informação em relação à classe *list* da biblioteca padrão do Python. Scikit-learn implementa diversos algoritmos de mineração de dados, bem como técnicas de transformação e pré-processamento. Scipy é uma biblioteca usada para computação científica. Matplotlib fornece funções para plotar imagens e gráficos. Por fim, librosa é uma biblioteca para manipulação de áudio. Também fornecemos junto com os arquivos a biblioteca `pysndfx`, que nada mais é do que um *wrapper* para o Sox. A biblioteca está disponível no Github.

```
In [1]: from IPython.display import display
```

0.3 2. Análise exploratória

A primeira tarefa é criar a estrutura de pastas necessárias para segmentar os *captchas*. Como cada *captcha* contém quatro caracteres, a ideia é gerar quatro arquivos `.wav`, um para cada caractere.

```
In [6]: from main import create_folder_structure
        create_folder_structure()
```

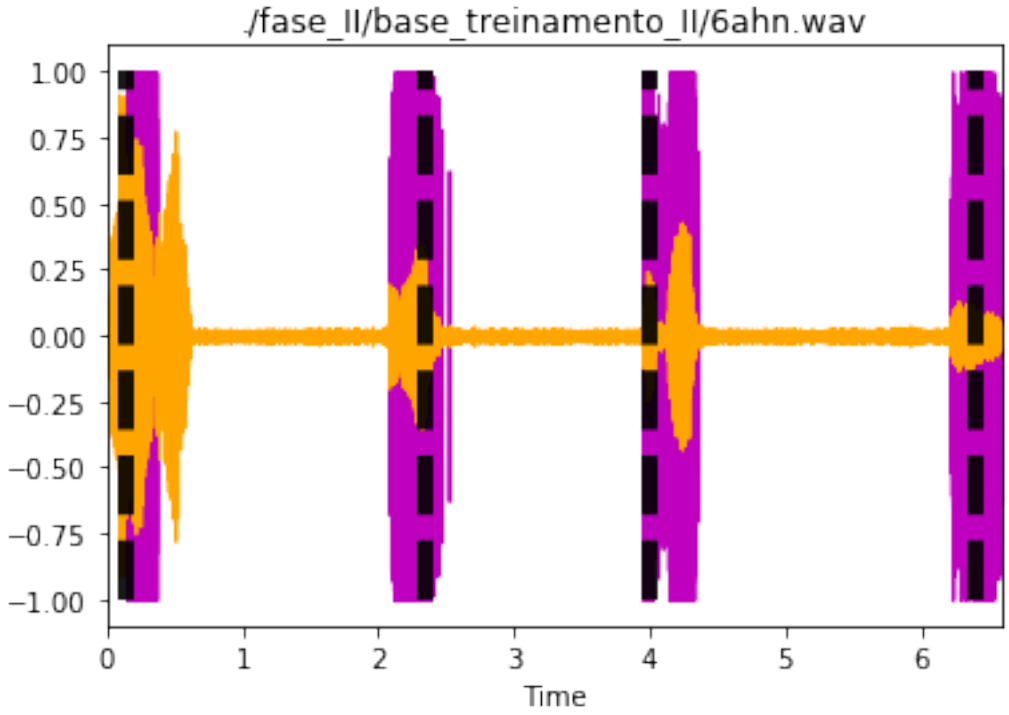
0.3.1 2.1 Segmentação

A segmentação segue os passos:

- O áudio é normalizado (tornando o maior pico = 1). Um *limiter* é aplicado com o parâmetro de 20dB. Assim, qualquer sinal com intensidade maior que este limite é diminuído até 20dB. Isso tenta reduzir a diferença entre picos altos e baixos, pois nem todas as letras são gravadas com a mesma intensidade;
- Frequências acima de 2500Hz e abaixo de 100Hz são descartadas, pois têm pouca informação vocal. Então, um equalizador é aplicado na faixa de 300Hz, intensificando essas frequências por 15dB. Na prática, isso causa um *boost* nas regiões onde há grande informação vocal, potencialmente até distorcendo-as, e deixando a diferença entre voz e fundo muito mais clara;
- Samples com intensidade menor que 0.5 são zerados, a fim de remover ruído de gravação. Idealmente, regiões silenciosas seriam zeradas;
- A função `find_peaks`, da biblioteca `scipy.signal` retorna todos os máximos e mínimos locais de um sinal, servindo para encontrar os picos do áudio. Mesmo passando parâmetros de altura mínima (de forma a apenas encontrar máximos) e distância mínima de 1s entre picos, por vezes a função encontra mais do que 4 picos, chamados daqui em diante de pontos;
- A intenção é que cada ponto represente um caractere. São escolhidos os pares de pontos mais próximos um do outro, e ambos são removidos e substituídos pela sua média. Isso é feito até que restem apenas 4 pontos. Teoricamente, ao fazer isso estamos pegando os caracteres com mais de um pico e colapsando suas posições até restar apenas um ponto, próximo ao seu centro;
- A partir de cada ponto encontrado, é extraída do áudio original uma região de 1s para cada lado, totalizando no máximo 2s de áudio. O ruído de fundo (silêncio) de cada lado e removido, isolando apenas a fala, e, como essa operação normalmente remove o começo e final do caractere, 0.25s são adicionados de volta de cada lado;
- Os áudios encontrados são exportados, assumindo que foram encontrados na ordem em que estão presentes no nome do arquivo.

Na imagem abaixo, desenhamos as ondas de um exemplo da base de treinamento. A onda roxa representa o áudio original; a onda laranja mostra o áudio após feito o processamento. Pode-se notar que os picos estão bem mais visíveis, e as regiões de ruído de fundo tornaram-se silêncio; dessa forma, temos um áudio próximo de "binarizado", onde 1 é fala, e 0 é silêncio. Os riscos em preto são os quatro pontos encontrados.

```
In [7]: import preprocessor as trim
        import importlib
        importlib.reload(trim)
        trim.trim('./fase_II/base_treinamento_II/6ahn.wav', '', True) #desenha o grafico
```



0.3.2 2.2 Atributos e análise preliminar de classificadores

```
In [2]: from model import train, individual_classifiers_results, test, get_final_model, final_te

In [3]: from display_results import resultados_acuracia, resultados_caracteres
```

A função `get_spectrum` extrai a intensidade média dentro de cada um dos espectros de áudio. Por exemplo, os espectros mais agudos poderiam ajudar um classificador a separar caracteres sibilantes (com sons de "S", como x e 7) de caracteres não sibilantes (como m e h).

Com o auxílio da biblioteca *librosa*, extraímos o MFCC (Mel-Frequency Cepstral Coefficients) das ondas sonoras. Para cada áudio, o MFCC retornado é uma matriz, onde cada linha possui um vetor numérico. O MFCC procura imitar as características que seres humanos percebem pelo ouvido. Para não lidar com um número absurdamente grande de atributos, usamos somente a mediana e o desvio padrão de cada vetor (40 ao todo), cujos valores foram tornados positivos. Isso foi feito porque os vetores do MFCC continham valores negativos e positivos (ondas), e assim a mediana tenderia a ser sempre próxima de zero.

Além disso, também com o auxílio da *librosa*, contamos quantas vezes o áudio corta o zero no eixo y, dividido pelo tamanho do áudio, a fim de ter um valor normalizado. Também extraímos o RMS dos áudios e usamos o desvio padrão, média e a mediana como atributo. O RMS funciona como uma média para os picos do áudio. Ainda com o auxílio da librosa, extraímos média e desvio padrão de *spectral_contrast*.

A função `extract_features` agrega os atributos acima citados em um vetor com 95 *features* ao todo. Os dados são normalizados de forma a terem média 0 e desvio 1. Isso é importante principalmente para algoritmos baseados em distância, de forma que atributos com valores maiores não

dominem a classificação em detrimento de outros cujos valores tendem a ser mais próximos de zero. Os resultados preliminares dos classificadores estão abaixo. Foi usada a base de treinamento fornecida para treino e a base de validação fornecida para teste.

```
In [4]: X_train, y_train, std_scale = train()

In [5]: captchas_svm, caracteres_svm, captchas_1nn, caracteres_1nn, captchas_3nn, caracteres_3nn

In [6]: display(resultados_acuracia("classificadores", [[("Acuracia SVM (captcha)", "{0:.2f}".fo
```

Métrica	Taxa
Acuracia SVM (captcha)	15.44%
Acuracia SVM (caracteres)	59.63%
Acuracia LDA (captcha)	13.42%
Acuracia LDA (caracteres)	60.98%
Acuracia 3NN (captcha)	3.36%
Acuracia 3NN (caracteres)	49.32%
Acuracia 1NN (captcha)	8.72%
Acuracia 1NN (caracteres)	54.05%

Acurácia classificadores

0.4 3. Metodologia

Para cada caso, é calculada a acurácia na detecção de caracteres e *captchas*.

Os algoritmos usados, conforme já mencionado, são 1NN, 3NN, LDA e SVM.

- **KNN**: O algoritmo recebe um elemento de teste e calcula sua distância para os demais elementos na base de dados. O elemento de teste recebe a classe do elemento mais próximo. No presente caso, foi usada a distância euclidiana, com valores de k igual a 1 e 3 (ver apêndice nos testes de K realizados).
- **SVM**: O *Support Vector Machine*, a partir de uma base de dados com elementos rotulados, constrói um modelo de pontos em um espaço de forma que cada classe esteja no espaço mais amplo possível. Então, ele traça retas, e os elementos são classificados conforme o local em que são colocados.
- **LDA**: O *Linear Discriminant Analysis* possui uma fronteira de decisão linear. O modelo é treinado por meio de de gaussianas geradas para cada classe.

Foi usada a base de treinamento fornecida para treino e a base de validação fornecida para teste. Os melhores resultados foram obtidos por LDA e SVM, conforme mostra a Seção 2. Então realizamos um *ensemble* usando maioria de votos. Como a acurácia de caracteres do LDA foi a mais alta, deixamos o seu resultado com peso 2 na contagem de votos, e os demais com peso 1.

0.5 4. Resultados

No apêndice (6.1) há ainda a acurácia na detecção de cada caracter individualmente

```
In [7]: accuracy_captcha, accuracy_character, wrong, correct, elements = test(X_train, y_train,
```

```
In [8]: display(resultados_acuracia("", [["Acuracia (captcha)", "{0:.2f}".format(accuracy_captc
```

Métrica	Taxa
Acuracia (captcha)	19.46%
Acuracia (caracteres)	63.85%

Acurácia A acurácia obtida na detecção de *captchas* não está muito distante do previsto, com base na acurácia individual de detecção de caracteres. A afirmação vem da suposição de que um classificador que obtenha 63.85% de acurácia para os caracteres tende, ao classificar um *captcha* com quatro caracteres, obteria:

$$0.6385 \cdot 0.6385 \cdot 0.6385 \cdot 0.6385 = 0.6385^4 = 16.62\%$$

Nossos testes chegaram a 19.46% de acerto.

A linha abaixo retorna o modelo final, que usa as bases de treino + validação para treinar um modelo a ser usado contra a base de teste.

```
In [23]: final_classifier, std_scale = get_final_model()
```

A linha abaixo executa o modelo

```
In [24]: accuracy_captcha, accuracy_character, wrong, correct, elements = final_test(std_scale,
```

```
In [ ]: display(resultados_caracteres("Resultados finais", correct, wrong, elements))
```

```
In [ ]: display(resultados_acuracia("", [("Acuracia (captcha)", "{0:.2f}".format(accuracy_captch
```

0.6 5. Comentários finais

- Dificuldades encontradas

  A segmentação não foi um processo trivial. Separar o áudio em faixas de tempo pré-determinadas era uma estratégia perigosa, e por isso foi necessário elaborar uma estratégia mais refinada.

- Ideias que não foram exploradas e a razão

0.7 6. Apêndice

0.7.1 6.1 Classificação caracteres por caractere de cada classificador

```
In [9]: display(resultados_caracteres("caracteres individuais LDA", corretos_lda, errados_lda, e
```

0.7.2 caracteres individuais LDA

Caractere	Acerto	Erro
6	42/70 (60.00)	28/70 (40.00)
7	53/63 (84.13)	10/63 (15.87)
a	45/64 (70.31)	19/64 (29.69)
b	24/52 (46.15)	28/52 (53.85)
c	31/66 (46.97)	35/66 (53.03)
d	24/54 (44.44)	30/54 (55.56)
h	50/55 (90.91)	5/55 (9.09)
m	25/56 (44.64)	31/56 (55.36)
n	19/45 (42.22)	26/45 (57.78)
x	48/67 (71.64)	19/67 (28.36)

```
In [10]: display(resultados_caracteres("caracteres individuais 3NN", corretos_3nn, errados_3nn,
```

0.7.3 caracteres individuais 3NN

Caractere	Acerto	Erro
6	45/70 (64.29)	25/70 (35.71)
7	37/63 (58.73)	26/63 (41.27)
a	35/64 (54.69)	29/64 (45.31)
b	28/52 (53.85)	24/52 (46.15)
c	24/66 (36.36)	42/66 (63.64)
d	13/54 (24.07)	41/54 (75.93)
h	44/55 (80.00)	11/55 (20.00)
m	22/56 (39.29)	34/56 (60.71)
n	16/45 (35.56)	29/45 (64.44)
x	28/67 (41.79)	39/67 (58.21)

```
In [11]: display(resultados_caracteres("caracteres individuais 1NN", corretos_1nn, errados_1nn,
```

0.7.4 caracteres individuais 1NN

Caractere	Acerto	Erro
6	39/70 (55.71)	31/70 (44.29)
7	38/63 (60.32)	25/63 (39.68)
a	40/64 (62.50)	24/64 (37.50)
b	19/52 (36.54)	33/52 (63.46)
c	28/66 (42.42)	38/66 (57.58)
d	19/54 (35.19)	35/54 (64.81)
h	45/55 (81.82)	10/55 (18.18)
m	30/56 (53.57)	26/56 (46.43)
n	25/45 (55.56)	20/45 (44.44)

Caractere	Acerto	Erro
x	37/67 (55.22)	30/67 (44.78)

In [12]: display(resultados_caracteres("caracteres individuais SVM", corretos_svm, errados_svm,

0.7.5 caracteres individuais SVM

Caractere	Acerto	Erro
6	42/70 (60.00)	28/70 (40.00)
7	52/63 (82.54)	11/63 (17.46)
a	46/64 (71.88)	18/64 (28.12)
b	29/52 (55.77)	23/52 (44.23)
c	29/66 (43.94)	37/66 (56.06)
d	8/54 (14.81)	46/54 (85.19)
h	50/55 (90.91)	5/55 (9.09)
m	30/56 (53.57)	26/56 (46.43)
n	18/45 (40.00)	27/45 (60.00)
x	49/67 (73.13)	18/67 (26.87)

In [13]: display(resultados_caracteres("Caracteres individuais ENSEMBLE", correct, wrong, elemen

0.7.6 Caracteres individuais ENSEMBLE

Caractere	Acerto	Erro
6	51/70 (72.86)	19/70 (27.14)
7	56/63 (88.89)	7/63 (11.11)
a	45/64 (70.31)	19/64 (29.69)
b	30/52 (57.69)	22/52 (42.31)
c	32/66 (48.48)	34/66 (51.52)
d	19/54 (35.19)	35/54 (64.81)
h	50/55 (90.91)	5/55 (9.09)
m	27/56 (48.21)	29/56 (51.79)
n	20/45 (44.44)	25/45 (55.56)
x	48/67 (71.64)	19/67 (28.36)

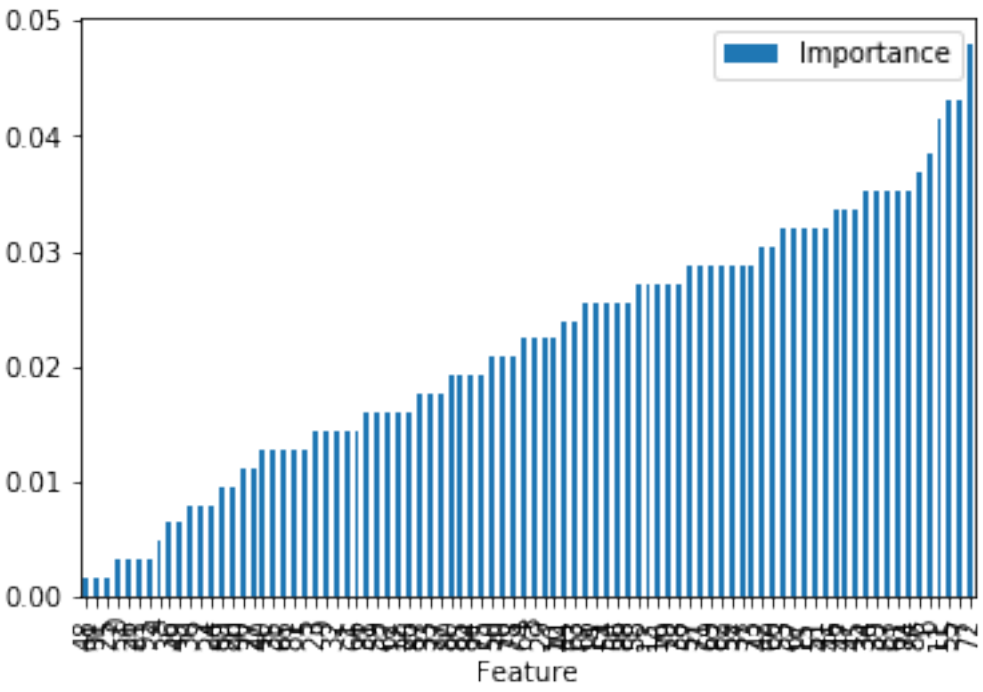
0.7.7 6.2 Importância dos atributos

```
In [27]: from sklearn.ensemble import RandomForestClassifier
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.base import clone
         import pandas as pd
         import numpy as np

         def dropcol_importances(rf, X_train, y_train):
```

```
    rf_ = clone(rf)
    rf_.random_state = 999
    rf_.fit(X_train, y_train)
    baseline = rf_.oob_score_
    imp = []
    for col in X_train.columns:
        X = X_train.drop(col, axis=1)
        rf_ = clone(rf)
        rf_.random_state = 999
        rf_.fit(X, y_train)
        o = rf_.oob_score_
        imp.append(baseline - o)
    imp = np.array(imp)
    I = pd.DataFrame(
            data={'Feature':X_train.columns,
                  'Importance':imp})
    I = I.set_index('Feature')
    I = I.sort_values('Importance', ascending=True)
    return I


rf = RandomForestClassifier(random_state=100, oob_score=True, n_estimators=50)
imp = dropcol_importances(rf, pd.DataFrame(X_train), y_train)
```

In [28]:
```
imp[imp.Importance > 0].plot.bar()
```

Out[28]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f81b09b6c18>
```



```
In [29]: most_important = imp[imp.Importance > 0].Importance.index

         most_important = imp[-44:].Importance.index

         X_raw = std_scale.inverse_transform(X_train)

         X_prime = X_raw[:, most_important]

In [30]: from sklearn.preprocessing import StandardScaler

         ss = StandardScaler()

         X_prime = ss.fit_transform(X_prime)

In [31]: from sklearn.model_selection import cross_val_score

         clf1 = cross_val_score(KNeighborsClassifier(n_neighbors=4), X_train, y_train, cv=10)

In [32]: clf2 = cross_val_score(KNeighborsClassifier(n_neighbors=4), X_prime, y_train,cv=10)
```

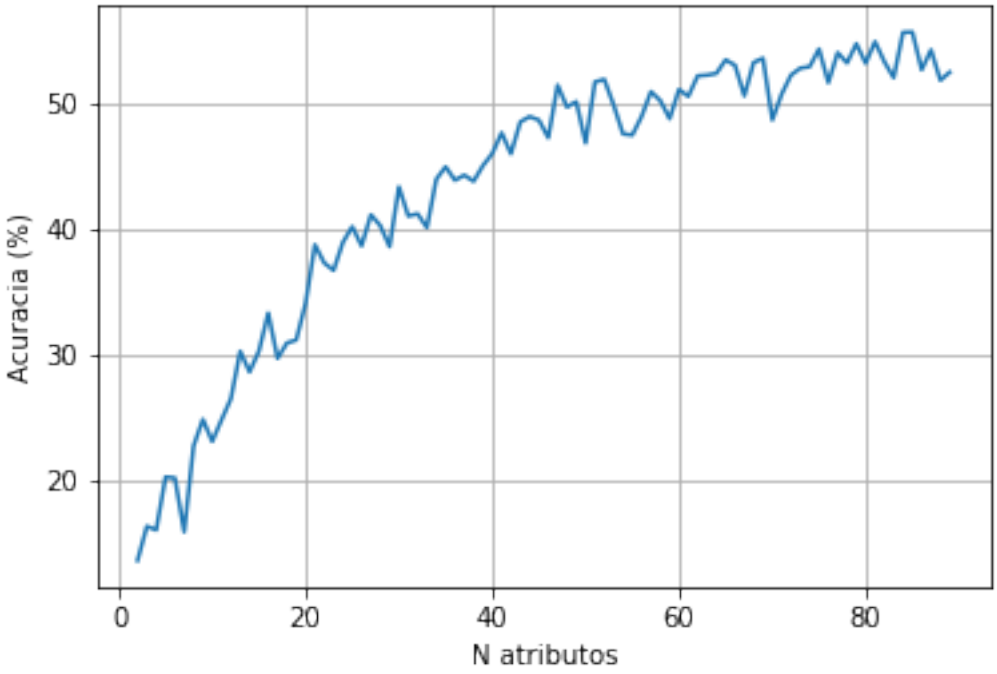
Acurácia do RF com 89 attrs

```
In [33]: "%.2f %%" % (clf1.mean() * 100)

Out[33]: '51.53 %'
```

Acurácia do RF com 44 attrs

```
In [34]: "%.2f %%" % (clf2.mean() * 100)

Out[34]: '53.11 %'

In [35]: most_important.shape

Out[35]: (44,)

In [36]: from sklearn.naive_bayes import GaussianNB
         from sklearn.linear_model import SGDClassifier
         from sklearn.tree import DecisionTreeClassifier

In [ ]: def calc_acc(n_features):
            most_important = imp[-n_features:].Importance.index
            X_prime = X_raw[:, most_important]
            ss = StandardScaler()
            X_prime = ss.fit_transform(X_prime)
            clf2 = cross_val_score(SGDClassifier(), X_prime, y_train,cv=10)
            return clf2.mean()

        graph = np.array([[i, calc_acc(i)] for i in range(89, 1, -1)])
```

```
In [38]: import matplotlib.pyplot as plt

         plt.plot(graph[:,0], graph[:,1] * 100)
         plt.xlabel(("N atributos"))
         plt.ylabel("Acuracia (%)")
         plt.grid()
```



Maior pico do gráfico:

```
In [39]: graph[graph[:,1].argmax()]

Out[39]: array([85.        ,  0.55626013])

In [40]: most_important[:9]

Out[40]: Int64Index([64, 3, 29, 10, 44, 63, 68, 14, 54], dtype='int64', name='Feature')
```

0.7.8 6.3 Acurácia do KNN, para diferentes valores de K

```
In [41]: def kNN(k):
             acc = cross_val_score(KNeighborsClassifier(n_neighbors=k), X_train, y_train,cv=10)
             return acc.mean()

         accs = np.array([[k,kNN(k)] for k in range(1, 15)] )

In [42]: accs[accs[:,1].argmax()]
```

```
Out[42]: array([1.        , 0.58121956])
```

```
In [43]: plt.plot(accs[:,0], accs[:,1] * 100)
         plt.xticks(range(1, 15))
         plt.grid()
         plt.xlabel("Número de vizinhos")
         plt.ylabel("Acurácia (%)")
```

```
Out[43]: Text(0,0.5,'Acurácia (%)')
```