Centro de Informática
Universidade Federal da Paraíba

Prova 2 de Inteligência artificial

Claudio de Souza Brito - Matrícula: 20170023696

João Pessoa, 2020.

Resumo

Esse relatório possui as metodologias usadas na resolução da segunda prova de inteligência artificial.

Sumário

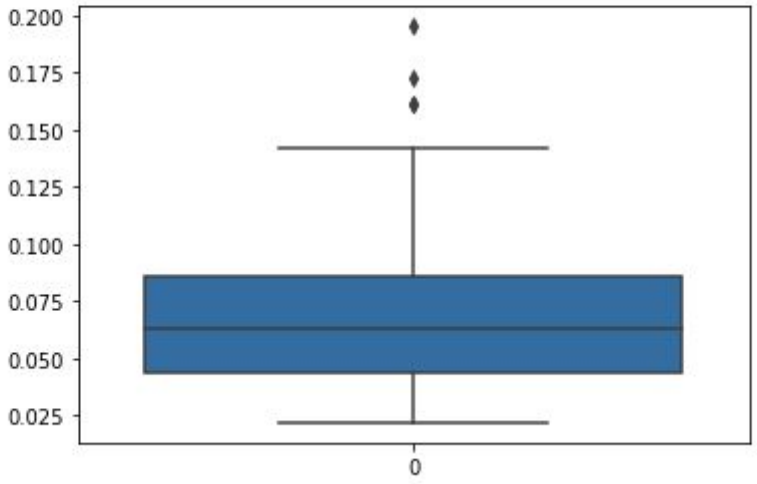
Questão 1.

1.1 Se livrando de outliers

Antes de começarmos a usar os métodos escolhidos, precisamos nos livrar de alguns outliers, valores que provavelmente foram erros ou ruídos, esses valores podem atrapalhar o julgamento dos métodos, por segurança vamos substituí-los. Vamos fazer isso indo em cada coluna a procura de valores acima do normal e substituindo pela mediana.

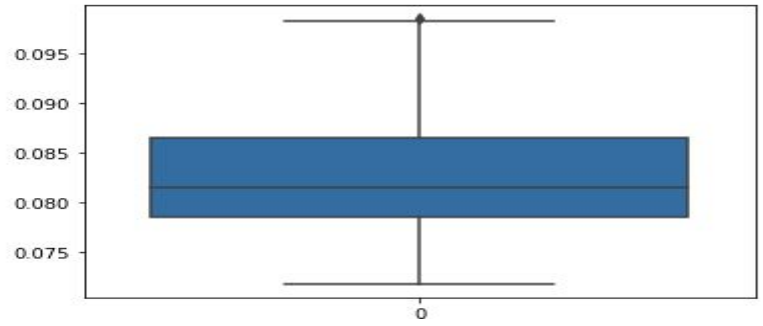
```
In [9]: df = pd.read_csv('questao1.csv', delimiter=",")
        df
        sns.boxplot(data=df["MM-centroid"])
        df["MM-centroid"].median()
```

```
Out[9]: 0.06287075
```



```
In [13]: df = pd.read_csv('questao1.csv', delimiter=",")
         df
         sns.boxplot(data=df["MM-flux"])
         df["MM-flux"].median()

Out[13]: 0.0815175
```



1.2 Normalização

A normalização é um recurso muito útil para manter nossos dados numa mesma faixa numérica, sem perder a coesão. É usado para comparar valores de colunas que têm escalas distintas. Houve uma tentativa de normalização, porém não foi possível continuar dessa forma pois o código teve vários erros, então seguimos sem:

```
In [18]: #NORMALIZACAO
         df_numeric = df[df.columns]
         # Normalize all of the numeric columns
         df_normalized = (df_numeric - df_numeric.mean()) / df_numeric.std()
```

1.3 Métodos de classificação

O primeiro método usado foi o K-NN, um método que faz inúmeros cálculos a entrada, e define se a entrada está mais perto de “A”, ou “B”. É um algoritmo muito simples de fazer, com uma proposta muito simples, então sempre é bem vindo, sem falar que foi feito para classificações binárias (0 ou 1), que é o nosso caso. O cálculo usado foi a distância euclidiana.

In [20]:

```
import random
from numpy.random import permutation
import math

#METODO KNN
# Embaralha aleatoriamente df
random_indices = permutation(df.index)

# teste = 20% das amostras
test_cutoff = math.floor(len(df)/5)

test = df.loc[random_indices[1:test_cutoff]]

# treino = resto (80%) das amostras
train = df.loc[random_indices[test_cutoff:]]
```

In [20]:

```
# colunas que vao servir para o treinamento
x_columns = ['BS1', 'BS2', 'BS3', 'HPA', 'HPB', 'LPA', 'LPB', 'MM-centroid',
       'MM-flux', 'MM-mfcc_0', 'MM-mfcc_1', 'MM-mfcc_10', 'MM-mfcc_11',
       'MM-mfcc_12', 'MM-mfcc_2', 'MM-mfcc_3', 'MM-mfcc_4', 'MM-mfcc_5',
       'MM-mfcc_6', 'MM-mfcc_7', 'MM-mfcc_8', 'MM-mfcc_9', 'MM-rolloff',
       'MS-centroid', 'MS-flux', 'MS-mfcc_1', 'MS-mfcc_10', 'MS-mfcc_11',
       'MS-mfcc_12', 'MS-mfcc_2', 'MS-mfcc_3', 'MS-mfcc_4', 'MS-mfcc_5',
       'MS-mfcc_6', 'MS-mfcc_7', 'MS-mfcc_8', 'MS-mfcc_9', 'MS-rolloff',
       'SM-centroid', 'SM-flux', 'SM-mfcc_0', 'SM-mfcc_1', 'SM-mfcc_10',
       'SM-mfcc_11', 'SM-mfcc_12', 'SM-mfcc_2', 'SM-mfcc_3', 'SM-mfcc_4',
       'SM-mfcc_5', 'SM-mfcc_6', 'SM-mfcc_7', 'SM-mfcc_8', 'SM-mfcc_9',
       'SM-rolloff', 'SS-centroid', 'SS-flux', 'SS-mfcc_0', 'SS-mfcc_1',
       'SS-mfcc_10', 'SS-mfcc_11', 'SS-mfcc_12', 'SS-mfcc_2', 'SS-mfcc_3',
       'SS-mfcc_4', 'SS-mfcc_5', 'SS-mfcc_6', 'SS-mfcc_7', 'SS-mfcc_8',
       'SS-mfcc_9', 'SS-rolloff', 'id']
# colnas que devem ser adivinhadas
y_column = ["amazed", "happy", "relaxing", "sad", "quiet", "angry"]

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)
# colocando o treino para funcionar
knn.fit(train[x_columns], train[y_column])
# adivinhando
predictions = knn.predict(test[x_columns])
```

In [22]:

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

# pegar as respostas para comparacao
actual = test[y_column]
print("\n acuracia K-NN")
mse = (((predictions - actual) ** 2).sum()) / len(predictions)
print( 1 - mse)
predictions = np.around(predictions)
```

O outro método foi o da árvore de decisão, um método um pouco mais complexo, que funciona para mais de duas classes. O algoritmo faz várias bifurcações até chegar na “folha” que é a classe em si. Esse método é bem fácil de entender e precisa de pouca preparação.

```
In [23]: #Importa biblioteca
         #Importa outras bibliotecas necessárias como pandas, numpy...
         from sklearn import tree
         #Assume que você tem X (preditor) e Y (alvo) para dados de treino e x_test(predictor) dos dados de teste
         # Cria o objeto tree
         model = tree.DecisionTreeClassifier(criterion='gini')
         # Para classificação, aqui você pode mudar o algoritmo para gini ou para entropy (Ganho de informação). Por default
         # model = tree.DecisionTreeRegressor() para regressão
         # Treina o modelo usando os dados de treino e de teste confere o score
         model.fit(train[x_columns], train[y_column])
         model.score(train[x_columns], train[y_column])
         #Prevê o resultado
         predicted= model.predict(test[x_columns])
         print("\n acuracia arvore de decisao")
         mse = (((predicted - actual) ** 2).sum()) / len(predicted)
         print( 1 - mse)
         predicted = np.around(predicted)
```

1.4 Métodos de análise de resultados

Foram escolhidos acurácia e matriz de confusão. A acurácia nada mais é do que 1 - a média dos erros obtidos no cálculo, quanto mais próximo a previsão do resultado final, menor a média de erros, e portanto maior a acurácia. Esse é o método mais fácil de ser implementado, e ele funciona razoavelmente bem, melhor quando a base de dados foi balanceada.

```
mse = (((predicted - actual) ** 2).sum()) / len(predicted)
print( 1 - mse)
```

A matriz de confusão é bastante útil, ela mostra exatamente o número de instâncias que foram classificadas corretamente ou não, é a maneira mais visual que existe.

```
predicted = np.around(predicted)
nactual = actual.to_numpy()
print("matriz de confusao amazed")
print(confusion_matrix(column(nactual, 0), column(predicted, 0)))
print("matriz de confusao happy")
print(confusion_matrix(column(nactual, 1), column(predicted, 1)))
print("matriz de confusao relaxing")
print(confusion_matrix(column(nactual, 2), column(predicted, 2)))
print("matriz de confusao sad")
print(confusion_matrix(column(nactual, 3), column(predicted, 3)))
print("matriz de confusao quiet")
print(confusion_matrix(column(nactual, 4), column(predicted, 4)))
print("matriz de confusao angry")
print(confusion_matrix(column(nactual, 5), column(predicted, 5)))
```

1.5 Discutindo resultados

Nessa sessão vamos falar sobre os resultados obtidos pelos métodos. Depois de retirar os outliers usamos k-nn e árvore de decisão:

acuracia K-NN
amazed 0.788718
happy 0.742564
relaxing 0.803077
sad 0.706667
quiet 0.778462
angry 0.739487
dtype: float64
matriz de confusao amazed
[[23 3]
[9 4]]
matriz de confusao happy
[[25 3]
[9 2]]
matriz de confusao relaxing
[[16 5]
[5 13]]
matriz de confusao sad
[[18 6]
[12 3]]
matriz de confusao quiet
[[23 5]
[9 2]]
matriz de confusao angry
[[20 8]
[9 2]]

acuracia arvore de decisao
amazed 0.820513
happy 0.641026
relaxing 0.717949
sad 0.717949
quiet 0.743590
angry 0.717949
dtype: float64
matriz de confusao amazed
[[25 1]
[6 7]]
matriz de confusao happy
[[22 6]
[8 3]]
matriz de confusao relaxing
[[15 6]
[5 13]]
matriz de confusao sad
[[18 6]
[5 10]]
matriz de confusao quiet

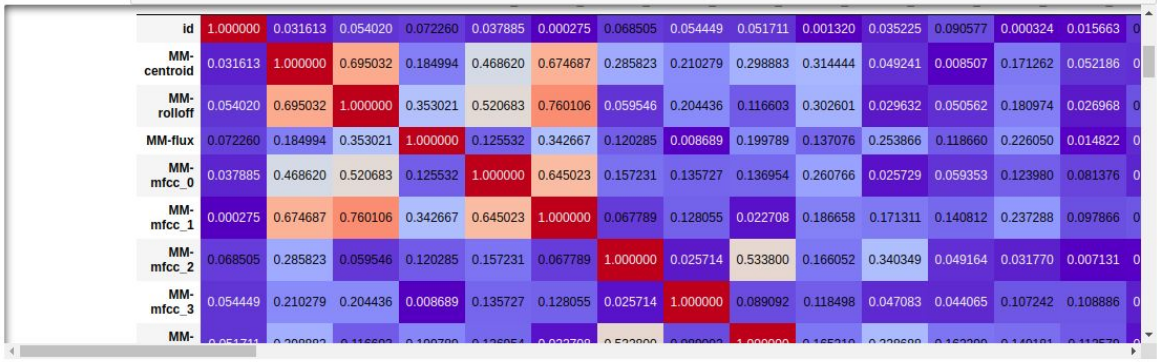
[[20 8]
 [2 9]]
matriz de confusao angry
[[24 4]
 [7 4]]

Como podemos ver a acurácia de ambos é bem parecida, mas na matriz de confusão vemos que o método do K-NN tem problemas em classificar a segunda classe (1), enquanto isso o método da árvore consegue mais sucesso do que falha.

Por questão de experimentação, vamos ver se retirando colunas com alta correlação é possível melhorar.

In [25]:
```
import numpy as np

# Create correlation matrix
corr_matrix = df.corr().abs()
corr_matrix.style.background_gradient(cmap = 'coolwarm')
```



	id	MM-centroid	MM-rolloff	MM-flux	MM-mfcc_0	MM-mfcc_1	MM-mfcc_2	MM-mfcc_3	MM-
id	1.000000	0.031613	0.054020	0.072260	0.037885	0.000275	0.068505	0.054449	
MM-centroid	0.031613	1.000000	0.695032	0.184994	0.468620	0.674687	0.285823	0.210279	
MM-rolloff	0.054020	0.695032	1.000000	0.353021	0.520683	0.760106	0.059546	0.204436	
MM-flux	0.072260	0.184994	0.353021	1.000000	0.125532	0.342667	0.120285	0.008689	
MM-mfcc_0	0.037885	0.468620	0.520683	0.125532	1.000000	0.645023	0.157231	0.135727	
MM-mfcc_1	0.000275	0.674687	0.760106	0.342667	0.645023	1.000000	0.067789	0.128055	
MM-mfcc_2	0.068505	0.285823	0.059546	0.120285	0.157231	0.067789	1.000000	0.025714	
MM-mfcc_3	0.054449	0.210279	0.204436	0.008689	0.135727	0.128055	0.025714	1.000000	
MM-									

In [26]:
```
#DROPANDO COLUNAS COM ALTA CORRELACAO
# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))

# Find features with correlation greater than 0.75
to_drop = [column for column in upper.columns if any(upper[column] > 0.75)]

# Drop features
df.drop(to_drop, axis=1, inplace=True)
```

Dropando todas as colunas com correlação maior que 75%.
Agora vamos rodar o código novamente.

K-NN
amazed   0.668
happy   0.694
relaxing   0.762
sad   0.758
quiet   0.772
angry   0.722
dtype: float64
matriz de confusao amazed

[[3 5]
 [5 7]]
matriz de confusao happy
[[8 3]
 [7 2]]
matriz de confusao relaxing
[[10 4]
 [4 2]]
matriz de confusao sad
[[13 2]
 [5 0]]
matriz de confusao quiet
[[15 0]
 [5 0]]
matriz de confusao angry
[[9 5]
 [5 1]]

acuracia arvore de decisao
amazed   0.666667
happy   0.641026
relaxing   0.717949
sad   0.666667
quiet   0.794872
angry   0.769231
dtype: float64
matriz de confusao amazed
[[21 3]
 [10 5]]
matriz de confusao happy
[[21 6]
 [8 4]]
matriz de confusao relaxing
[[18 1]
 [10 10]]
matriz de confusao sad
[[18 7]
 [6 8]]
matriz de confusao quiet
[[26 5]
 [3 5]]
matriz de confusao angry
[[22 5]
 [4 8]]

Como podemos ver, a acurácia geral nos dois métodos diminuiu, e além disso as matrizes de confusão da árvore tiveram resultados amargos. Infelizmente a correlação não ajudou, então voltamos ao estado original.

A próxima ideia foi balancear as classes das seis emoções, vemos que tem bem mais 0 do que 1.

```
0    148
1     54
Name: amazed, dtype: int64
0    143
1     59
Name: happy, dtype: int64
0    106
1     96
Name: relaxing, dtype: int64
0    129
1     73
Name: sad, dtype: int64
0    143
1     59
Name: quiet, dtype: int64
0    144
1     58
Name: angry, dtype: int64
```

```
In [81]: df = pd.read_csv('questao1.csv', delimiter=",")
         df
         from sklearn.utils import resample
         # Separate majority and minority classes
         df_majority = df[df.quiet==0]
         df_minority = df[df.quiet==1]

         # Downsample majority class
         df_majority_downsampled = resample(df_majority,
                                            replace=False,    # sample without replacement
                                            n_samples=df['quiet'].value_counts()[1],     # to match minority class
                                            random_state=123) # reproducible results

         # Combine minority class with downsampled majority class
         df_downsampled = pd.concat([df_majority_downsampled, df_minority])

         # Display new class counts
         df_downsampled.quiet.value_counts()
         # 1    49
         # 0    49
         # Name: balance, dtype: int64
```

Eliminamos algumas instâncias com base na emoção “quiet”:

```
K-NN
amazed      0.736364
happy       0.790909
relaxing    0.698182
sad         0.680000
quiet       0.650909
angry       0.805455
```

dtype: float64
matriz de confusao amazed
[[15 1]
 [6 0]]
matriz de confusao happy
[[17 0]
 [5 0]]
matriz de confusao relaxing
[[5 4]
 [9 4]]
matriz de confusao sad
[[2 6]
 [6 8]]
matriz de confusao quiet
[[4 7]
 [7 4]]
matriz de confusao angry
[[15 2]
 [4 1]]

acuracia arvore de decisao
amazed   0.772727
happy   0.727273
relaxing   0.727273
sad   0.590909
quiet   0.636364
angry   0.772727
dtype: float64
matriz de confusao amazed
[[14 2]
 [3 3]]
matriz de confusao happy
[[15 2]
 [4 1]]
matriz de confusao relaxing
[[6 3]
 [3 10]]
matriz de confusao sad
[[5 3]
 [6 8]]
matriz de confusao quiet
[[7 4]
 [4 7]]
matriz de confusao angry
[[14 3]
 [2 3]]

Mais uma vez tivemos acurácias baixas, as matrizes de confusão do k-NN continuam sem saber classificar a segunda classe, mas as matrizes da árvore continuam bem. Indo mais além, não satisfeito, foi realizado mais um balanceamento, dessa vez com base na emoção “happy”.

```
K-NN
amazed      0.720
happy       0.670
relaxing    0.675
sad         0.850
quiet       0.720
angry       0.845
dtype: float64
matriz de confusao amazed
[[4 0]
 [3 1]]
matriz de confusao happy
[[2 1]
 [5 0]]
matriz de confusao relaxing
[[0 4]
 [1 3]]
matriz de confusao sad
[[6 1]
 [1 0]]
matriz de confusao quiet
[[4 2]
 [2 0]]
matriz de confusao angry
[[6 1]
 [1 0]]

acuracia arvore de decisao
amazed      0.625
happy       0.750
relaxing    0.375
sad         1.000
quiet       1.000
angry       0.625
dtype: float64
matriz de confusao amazed
[[2 2]
 [1 3]]
matriz de confusao happy
[[2 1]
 [1 4]]
matriz de confusao relaxing
```

[[2 2]
 [3 1]]
matriz de confusao sad
[[7 0]
 [0 1]]
matriz de confusao quiet
[[6 0]
 [0 2]]
matriz de confusao angry
[[5 2]
 [1 0]]

O K-NN acabou com acurácias regulares, não tão ruins quanto a original, mostrando que é um método que se mantêm bem constante com a retirada de instâncias de treino e teste. Mas terminou sem saber classificar a segunda classe. O método da árvore Bem pelo contrário, suas matrizes de confusão parecem estar bem no geral, mas ao olhar para a acurácia vemos um fenômeno bizarro, duas emoções foram perfeitamente classificadas (note que nenhum dos balanceamentos usou a emoção "sad" como base), porém as outras emoções tiveram resultados medíocres, e "relaxing" um resultado péssimo, mostrando que a árvores de decisões é um método que se torna caótico ao longo que perde informações(instâncias) para treino e teste

Questão 2.

2.1 Pré Processamento clássico

A primeira coisa que faremos é caçar valores nulos:

```
In [2]: df.isnull().sum()
Out[2]: holiday              0
        temp                 0
        rain_1h              0
        snow_1h              0
        clouds_all           0
        weather_main         0
        weather_description  0
        date_time            0
        traffic_volume       0
        dtype: int64
```

Após isso vamos eliminar as linhas duplicadas:

```
dtype: int64

In [3]: print(df.duplicated().value_counts())
        df.drop_duplicates(inplace=True)
        print(df)
```

Verificar a matriz de correlação, e eliminar coluna que tenha mais de 0,75 de correlação com outra:

```
In [4]: corr_matrix = df.corr().abs()
        corr_matrix.style.background_gradient(cmap = 'coolwarm')
```

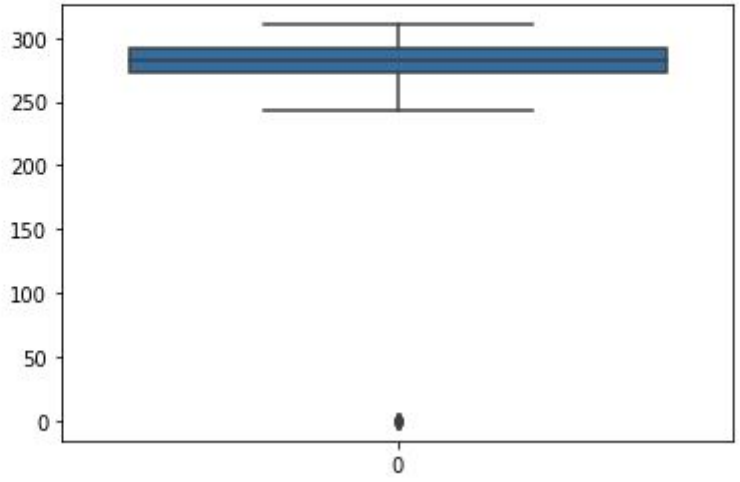
Out[4]:

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
temp	1.000000	0.009070	0.019756	0.101968	0.130161
rain_1h	0.009070	1.000000	0.000090	0.004818	0.004715
snow_1h	0.019756	0.000090	1.000000	0.027934	0.000736
clouds_all	0.101968	0.004818	0.027934	1.000000	0.067138
traffic_volume	0.130161	0.004715	0.000736	0.067138	1.000000

Retirando outliers absurdos:

```
In [5]: sns.boxplot(data=df["temp"])
        df["temp"].median()
```

Out[5]: 282.45



Algumas colunas da base de dados parecem ser bastante importantes, mas não são valores numéricos, se quisermos usa-los na métrica de regressão, precisamos quantificá-los:

In [13]:
```
for i in df['holiday']:
    df['holiday'].replace(to_replace = 'None', value = 0, inplace=True)
    df['holiday'].replace(to_replace = 'Labor Day', value = 1, inplace=True)
    df['holiday'].replace(to_replace = 'Martin Luther King Jr Day', value = 2, inplace=True)
    df['holiday'].replace(to_replace = 'Christmas Day', value = 3, inplace=True)
    df['holiday'].replace(to_replace = 'Thanksgiving Day', value = 4, inplace=True)
    df['holiday'].replace(to_replace = 'New Years Day', value = 5, inplace=True)
    df['holiday'].replace(to_replace = 'Washingtons Birthday', value = 6, inplace=True)
    df['holiday'].replace(to_replace = 'Memorial Day', value = 7, inplace=True)
    df['holiday'].replace(to_replace = 'Independence Day', value = 8, inplace=True)
    df['holiday'].replace(to_replace = 'State Fair', value = 9, inplace=True)
    df['holiday'].replace(to_replace = 'Columbus Day', value = 10, inplace=True)
    df['holiday'].replace(to_replace = 'Veterans Day', value = 11, inplace=True)
```

In [14]:
```
for i in df['weather_main']:
    df['weather_main'].replace(to_replace = 'Clouds', value = 0, inplace=True)
    df['weather_main'].replace(to_replace = 'Clear', value = 1, inplace=True)
    df['weather_main'].replace(to_replace = 'Mist', value = 2, inplace=True)
    df['weather_main'].replace(to_replace = 'Rain', value = 3, inplace=True)
    df['weather_main'].replace(to_replace = 'Snow', value = 4, inplace=True)
    df['weather_main'].replace(to_replace = 'Drizzle', value = 5, inplace=True)
    df['weather_main'].replace(to_replace = 'Haze', value = 6, inplace=True)
    df['weather_main'].replace(to_replace = 'Thunderstorm', value = 7, inplace=True)
    df['weather_main'].replace(to_replace = 'Fog', value = 8, inplace=True)
    df['weather_main'].replace(to_replace = 'Smoke', value = 9, inplace=True)
    df['weather_main'].replace(to_replace = 'Squall', value = 10, inplace=True)
```

In [15]:
```
for i in df['weather_description']:
    df['weather_description'].replace(to_replace = 'sky is clear', value = 0, inplace=True)
    df['weather_description'].replace(to_replace = 'mist', value = 1, inplace=True)
    df['weather_description'].replace(to_replace = 'overcast clouds', value = 2, inplace=True)
    df['weather_description'].replace(to_replace = 'broken clouds', value = 3, inplace=True)
    df['weather_description'].replace(to_replace = 'scattered clouds', value = 4, inplace=True)
    df['weather_description'].replace(to_replace = 'light rain', value = 5, inplace=True)
    df['weather_description'].replace(to_replace = 'few clouds', value = 6, inplace=True)
    df['weather_description'].replace(to_replace = 'light snow', value = 7, inplace=True)
    df['weather_description'].replace(to_replace = 'Sky is Clear', value = 8, inplace=True)
    df['weather_description'].replace(to_replace = 'moderate rain', value = 9, inplace=True)
    df['weather_description'].replace(to_replace = 'haze', value = 10, inplace=True)
    df['weather_description'].replace(to_replace = 'light intensity drizzle', value = 11, inplace=True)
    df['weather_description'].replace(to_replace = 'fog', value = 12, inplace=True)
    df['weather_description'].replace(to_replace = 'proximity thunderstorm', value = 13, inplace=True)
    df['weather_description'].replace(to_replace = 'drizzle', value = 14, inplace=True)
    df['weather_description'].replace(to_replace = 'heavy snow', value = 15, inplace=True)
    df['weather_description'].replace(to_replace = 'heavy intensity rain', value = 16, inplace=True)
    df['weather_description'].replace(to_replace = 'snow', value = 17, inplace=True)
    df['weather_description'].replace(to_replace = 'proximity shower rain', value = 18, inplace=True)
    df['weather_description'].replace(to_replace = 'thunderstorm', value = 19, inplace=True)
    df['weather_description'].replace(to_replace = 'heavy intensity drizzle', value = 20, inplace=True)
    df['weather_description'].replace(to_replace = 'thunderstorm with heavy rain', value = 21, inplace=True)
    df['weather_description'].replace(to_replace = 'thunderstorm with light rain', value = 22, inplace=True)
    df['weather_description'].replace(to_replace = 'proximity thunderstorm with rain', value = 23, inplace=True)
    df['weather_description'].replace(to_replace = 'thunderstorm with rain', value = 24, inplace=True)
    df['weather_description'].replace(to_replace = 'smoke', value = 25, inplace=True)
```

2.2 Normalização

Agora normalizamos a base de dados para termos uma só escala:

```
In [17]:
#NORMALIZACAO
col = ['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday','traffic_volume
df_numeric = df[col]
# Normalize all of the numeric columns
df_normalized = (df_numeric - df_numeric.mean()) / df_numeric.std()
dfbu = df
df = df_normalized
```

2.3 Regressão

Nosso primeiro método é múltipla regressão linear:

```
In [26]:
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
X = df[['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday']].values
y = df['traffic_volume'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
regressor = LinearRegression()
regressor.fit(X_train, y_train)

Out[26]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [19]:
y_pred = regressor.predict(X_test)
mse = (((y_pred - y_test) ** 2).sum()) / len(y_pred)
mse

Out[19]: 0.9650049503192444
```

Nosso próximo método foi regressão polinomial, um método muito útil que é usado quando o cientista de dados percebe que a relação entre os dados não é linear:

```
In [27]:
from sklearn.preprocessing import PolynomialFeatures
X_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_, y, test_size=0.3, random_state=0)
model = LinearRegression()
model.fit(X_train, y_train)

Out[27]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [21]:
y_pred = model.predict(X_test)
mse = (((y_pred - y_test) ** 2).sum()) / len(y_pred)
mse

Out[21]: 0.9571187731696689
```

Como podemos ver esses são péssimos resultados, mesmo com 48 mil instâncias, dados normalizados, dados quantificados a previsão é quase inútil, a regressão polinomial parece ser levemente melhor. Não entendi porque foi um resultado tão ruim.

2.4 Validação cruzada

Em uma tentativa de melhorar nosso resultados, a validação cruzada oferece novas oportunidades ao mexer as instâncias entre treino e teste.

In [23]:
```
from sklearn.model_selection import KFold
kf = KFold(n_splits = 10, shuffle = True, random_state = 2)
for result in kf.split(df):
    train = df.iloc[result[0]]
    test =  df.iloc[result[1]]
    X_train = train[['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday']]
    y_train = train['traffic_volume'].values
    X_test = test[['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday']].va
    y_test = test['traffic_volume'].values
    regressor = LinearRegression()
    regressor.fit(X_train, y_train)
    y_pred = regressor.predict(X_test)
    mse = (((y_pred - y_test) ** 2).sum()) / len(y_pred)
    print(mse)
```

```
0.9623098078824827
0.9656256237591683
0.9542628767187414
0.9594479096968328
0.956673489091355
0.9856812621579275
0.9520967416697148
0.9672976848427485
0.9474721284682888
0.9862841807425324
```

In [24]:
```
kf = KFold(n_splits = 10, shuffle = True, random_state = 2)
for result in kf.split(df):
    train = df.iloc[result[0]]
    test =  df.iloc[result[1]]
    X_train = train[['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday']]
    y_train = train['traffic_volume'].values
    X_test = test[['temp', 'rain_1h', 'snow_1h', 'clouds_all', 'weather_description', 'weather_main', 'holiday']].va
    y_test = test['traffic_volume'].values
    X_train_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(X_train)
    X_test_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(X_test)
    regressor = LinearRegression()
    regressor.fit(X_train_, y_train)
    y_pred = regressor.predict(X_test_)
    mse = (((y_pred - y_test) ** 2).sum()) / len(y_pred)
    print(mse)
```

```
0.9476167022484642
0.9547193543878264
0.9423320702378434
0.9493070290675358
0.9490648017891293
0.9711930591991383
0.9377106054444985
0.9590007931738508
0.9366447947696415
0.9760601056362037
```

Não houve sucesso. Rodar duas vezes também não teve sucesso.

Questão 3.

3.1 Pré processamento básico

Começamos eliminando valores nulos e duplicados:

```
In [2]: df.isnull().sum()
```

```
Out[2]: status_id              0
        status_type            0
        status_published       0
        num_reactions          0
        num_comments           0
        num_shares             0
        num_likes              0
        num_loves              0
        num_wows               0
        num_hahas              0
        num_sads               0
        num_angrys             0
        Column1             7050
        Column2             7050
        Column3             7050
        Column4             7050
        dtype: int64
```

```
In [3]: df.drop(['Column1','Column2','Column3','Column4'],axis=1, inplace=True)
```

```
In [4]: df.duplicated()
        df.drop_duplicates(keep = False, inplace = True)
        df
```

Depois vemos a matriz de correlação e eliminamos colunas que tenham mais de 75% de correlação:

```
In [5]: corr_matrix = df.corr().abs()
        corr_matrix.style.background_gradient(cmap = 'coolwarm')
```

Out[5]:

	num_reactions	num_comments	num_shares	num_likes	num_loves	num_wows	num_hahas	num_sads	num_angrys
num_reactions	1.000000	0.161976	0.269280	0.994392	0.316847	0.255359	0.183631	0.090961	0.150136
num_comments	0.161976	1.000000	0.640432	0.110539	0.521416	0.164640	0.325079	0.284761	0.256788
num_shares	0.269280	0.640432	1.000000	0.187432	0.820569	0.412019	0.399902	0.242637	0.356150
num_likes	0.994392	0.110539	0.187432	1.000000	0.216508	0.191800	0.125721	0.063348	0.106857
num_loves	0.316847	0.521416	0.820569	0.216508	1.000000	0.511524	0.507967	0.251003	0.423345
num_wows	0.255359	0.164640	0.412019	0.191800	0.511524	1.000000	0.289194	0.097945	0.202104
num_hahas	0.183631	0.325079	0.399902	0.125721	0.507967	0.289194	1.000000	0.170304	0.240888
num_sads	0.090961	0.284761	0.242637	0.063348	0.251003	0.097945	0.170304	1.000000	0.145951
num_angrys	0.150136	0.256788	0.356150	0.106857	0.423345	0.202104	0.240888	0.145951	1.000000

```
In [6]: # Select upper triangle of correlation matrix
        upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))

        # Find features with correlation greater than 0.75
        to_drop = [column for column in upper.columns if any(upper[column] > 0.75)]

        # Drop features
        df.drop(to_drop, axis=1, inplace=True)
```

Depois disso nós vamos quantificar valores de status_type, faz sentido usar esses dados na clusterização pois eles têm informação relevante em classificar as mídias. Por outro lado vamos eliminar a coluna da data, pois é muito difícil quantificar essa coluna e também não parece ser tão relevante para agrupamento:

In [10]:
```
for i in df['status_type']:
    df['status_type'].replace(to_replace = 'photo', value = 0, inplace=True)
    df['status_type'].replace(to_replace = 'video', value = 1, inplace=True)
    df['status_type'].replace(to_replace = 'status', value = 2, inplace=True)
    df['status_type'].replace(to_replace = 'link', value = 3, inplace=True)
```

In [11]:
```
print(df['status_type'].value_counts())
```

```
0    4200
1    2332
2     353
3      63
Name: status_type, dtype: int64
```

In [12]:
```
df.drop(['status_published'], axis=1, inplace=True)
print(df.duplicated().value_counts())
df.drop_duplicates(inplace=True)
print(df)
```

3.2 Normalização

In [14]:
```
#NORMALIZACAO
col = ['status_type', 'num_reactions', 'num_comments', 'num_shares', 'num_wows', 'num_hahas', 'num_sads', 'num_angr
df_numeric = df[col]
# Normalize all of the numeric columns
df_normalized = (df_numeric - df_numeric.mean()) / df_numeric.std()
dfbu = df
df_normalized
```

3.3 PCA

PCA é uma técnica de reduzir o número de atributos(colunas) resumindo o número de dados sem perder muita informação:

In [15]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=5)
principalComponents = pca.fit_transform(df_normalized)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2', 'principal component 3', 'principal com
principalDf
```

3.4 Clusterização

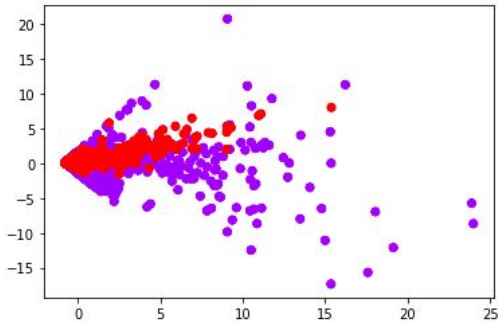
Nosso primeiro método é a clusterização hierárquica “single”:

```
In [37]: from sklearn.cluster import AgglomerativeClustering
         import numpy as np
         clustering = AgglomerativeClustering().fit(df)
         AgglomerativeClustering(n_clusters=8, linkage = 'single')
         clustering.labels_

Out[37]: array([0, 0, 0, ..., 1, 1, 1])
```

```
In [38]: import matplotlib.pyplot as plt
         plt.scatter(df.iloc[:,0], df.iloc[:,1], c=clustering.labels_, cmap='rainbow')

Out[38]: <matplotlib.collections.PathCollection at 0x7fbae151b668>
```



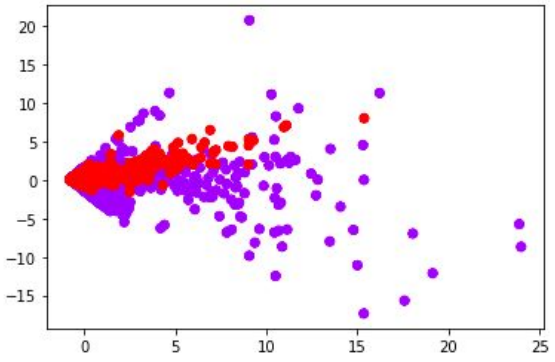
Passando agora para o “complete”:

```
In [39]: clustering = AgglomerativeClustering().fit(df)
         AgglomerativeClustering(n_clusters=8, linkage = 'complete')
         clustering.labels_

Out[39]: array([0, 0, 0, ..., 1, 1, 1])

In [40]: plt.scatter(df.iloc[:,0], df.iloc[:,1], c=clustering.labels_, cmap='rainbow')

Out[40]: <matplotlib.collections.PathCollection at 0x7fbae12dc2e8>
```

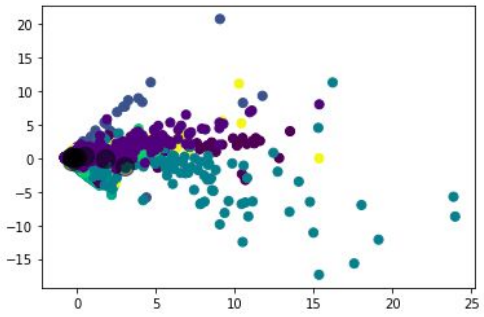


Como podemos ver não houve quase nenhuma diferença, além disso os métodos parecem não ter ido muito bem.

Agora vamos ver como fica o K-means com todos os centroids gerados aleatoriamente:

In [36]:
```
plt.scatter(df.iloc[:, 0], df.iloc[:, 1], c=kmeans.predict(df), s=50, cmap='viridis')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```



Como podemos ver, o gráfico é bem parecido com aquele gerado pelo hierarquico, mas é bem mais diversificado