Documentación

Machine Learning para sistemas inteligentes
Irazoqui - Golubei

Siendo honestos, no tuvimos demasiado tiempo por diferentes razones, asi que disculpen un poco el orden que puede haber. Hicimos lo mejor para ordenar un poco los experimentos que hicimos, en que momento, y sobre que eran. Intentamos detallar debajo del titulo para cada experimento que notebook contiene ese experimento.

Link al repo con todas las cosas que hicimos

Análisis básico

Para empezar hicimos un análisis básico del problema, con un preprocesamiento básico de los datos del dataframe. Utilizamos un árbol simple como modelo de aprendizaje, y más alla de borrar la columna duplicada Id, no hubo más preprocesamiento aunque hubieran palabras claramente sin sentido (por ejemplo) en los datos.

Lo hicimos de esta forma para tener una idea clara inicial de cual es la exactitud de un análisis básico del problema.

Esto se encuentra en `Procesamiento basico.ipynb`.

Preprocesamiento inicial

El siguiente paso es investigar algunas de las herramientas básicas de preprocesamiento: stop words, lematizing, stemming, etc.

En esta ocasión vamos a utilizar nuevamente el árbol para comparar con los datos iniciales.

Este paso lo vamos a repetir luego de tener el preprocesamiento inicial, pero cuando estemos utilizando una red neuronal.

→ Primero intentamos buscar el mejor vectorizer.
→ Luego pasamos a buscar la mejor red cambiando la arquitectura (capas, regularizadores)

→ Hecho eso hay que probar los métodos de ensemble como Random Forest, Boosting, Bagging, etc.

Lemmatizing, puntuación, stopwords

Se encuentra en: Arbol-lematizing,stemming,stopwords.ipynb

Para seguir procesando los datos, empezamos usando un lemmatizer. Esto elimino todas los strings sin sentido de los datos, y muchas de las palabras.

Un problema que surgió al usar el lemmatizer, es que tomaba la puntuación como palabras diferentes, es decir “analyze” != “analyze,” != “analyze:”. Esto no tiene sentido, por lo que primero borramos algunos símbolos que no queremos.

```
def is_Not_Link(word):
  return word[0:4] != "http"

def remove_Punctuation(doc_split):
  for i in range(len(doc_split)):
    word = doc_split[i];
    if is_Not_Link(word):
      for elements in word:
        if elements in All_punct:
          doc_split[i] = word.replace(elements, "")
  return doc_split
```

Decidimos no eliminar los links al principio

Luego usamos las stopwords:

```
def remove_Stopwords(doc_split):
  doc_with_Stopwords = doc_split.copy();
  for i in range(len(doc_split)):
    word = doc_split[i];
    if word in nltk.corpus.stopwords.words('english'):
      doc_with_Stopwords.remove(word);

  return doc_with_Stopwords
```

Todo esto se lo pasamos al CountVectorizer como un nuevo analyzer:

```
def new_analyzer(doc):
  doc_split = doc.split();
  doc_split = remove_Punctuation(doc_split);
  doc_split = remove_Stopwords(doc_split);
  return (Lemmatizer.lemmatize(w.lower()) for w in doc_split);
```

```
vectorizer_stop_words_stemmed = CountVectorizer(stop_words =
nltk.corpus.stopwords.words('english'), analyzer=new_analyzer)
```

Árbol

Luego de hacer estos pasos, se creo otro árbol y se uso para predecir Y. Sin embargo, la accuracy era casi igual (un poco peor aveces incluso)

Por lo tanto, decidimos pasar a usar redes neuronales

Red Neuronal

Empezamos por crear una red neuronal extremadamente simple para probar, y luego la mejoramos un poco:

```
adam_optimizer='Adam'
loss='binary_crossentropy'
metric='accuracy'

model = Sequential()
model.add(Dense(20, input_shape=(bag_of_words.shape[1],)))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dense(30, use_bias=True))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dense(1, use_bias=True))
model.add(Activation('sigmoid'))
model.compile(loss=loss, optimizer=adam_optimizer, metrics=[metric])
```

Accuracy:

```
0.9141555711282411
```

Esta es una accuracy mucho mejor que la que vimos con el árbol

K-Fold

Luego de crear la red neuronal me dispuse a crear un método para hacer k-fold, dado que los proximos pasos van a ser de ingeniería de atributos y hacer experimentos, me gustaría tener una estimación de la accuracy acertada. Para eso exactamente es que k-fold sirve
Sin embargo, por alguna razón K-Fold me esta dando accuracy mucho mayor que si lo hiciera normal sin K-fold. No estoy seguro por que es, pero como lo hice a mano quizas me equivoque en algo.

Claramente no esta bien, ya que el error ronda en 0.91…tanto sin k-fold como en test, pero al hacer k-fold, salta a .96 o numeros mucho mayores.

Una lastima no poder aplicarlo bien y tener información acertada sobre los cambios en los modelos y datos, pero tampoco tengo el tiempo para arreglarlo.

Experimentos - Datos

Borrar los links

Experimento-BorrarLinks.ipynb

Tanto en k-fold como sin usarla, la accuracy fue peor

```
0.9078486334968465
```

K-fold

```
0.9601903976312638
```

No voy a usar mas k-fold porque no parece estar funcionando bien

Borrar los @

Experimento-Borrar@.ipynb
Ocurrio lo mismo que con los links

Borrar los

Experimento-BorrarHashtags.ipynb

Borrando los hashtag:

```
0.8959355290819901
```

Borrando el símbolo:

```
0.5073580939032937
```

Emojis y simbolso basura

IngAtributos-Emojis.ipynb

Revisando los feature names, nos dimos cuenta de que habian muchos emojis que previenen que algunas palabras se cuenten bien. Vamos a separarlos a ver que pasa. De paso, borraremos algunos simbolos que son basura (Los vamos a mezclar con los de puntuación por performance).

Borrar los emoji en el analyzer disminuyo el accuracy.

```
def remove_Emojis(doc_split):
  i = 0
  while i < len(doc_split):
    word = doc_split[i];
    for elements in word:
      if elements in Emojis:
        emoji = elements
        doc_split[i] = word.replace(elements, "")
        doc_split.append(doc_split[i])
        doc_split[i] = emoji
    i = i + 1;
  return doc_split
```

Sin embargo, contar los emojis y agregar el resultado en otra columna, subio el accuracy:

```
emojis =
"ℹ️↗️⏺️▪️◾▶️☎️☑️☺️⚖️⛪✅✈️✉️✔️❄️❌❌❗❤️➡️➡️🔜🆕🆙🌍🌐🏠🏪🏥🏫👇👉👉🏽👨👩
⚕️💉💊📃📈📉📍📏📢📨📺🔰🔴🔹🕌🕐😷🙏🚶♀️🛡️🟢🤔🤱🥼🧤🧪"
def contarEmojis(row):
  count = 0;
  for letter in row['tweet']:
    if letter in emojis:
      count = count + 1
  return count
```

```
0.9323756131744919
```

Por alguna razon, quitar el simbolo 🔸 hace que perdamos accuracy.

TF-IDF Vectorizer

RedNeuronal-TFIDF.ipynb
Vamos a pasar a usar un modelo TFIDF como pide la letra antes de hacer la ingeniería de atributos, así no hay que repetirlo.

El accuracy fue:

```
0.9180098107918 71
```

Casi 0.92, una mejora considerable a este punto.

Experimento Max_Features

Experimento-Max_Features.ipynb
Para este experimento probamos cambiar la variable de max_features. Si lo dejamos sin poner un máximo, llegamos a una accuracy de:

```
0.9330763840224247
```

(Esto se hizo luego de hacer el experimento con el nuevo atributo "Cantidad de letras"). Se ve un aumento de la accuracy considerablde del 1%. Sin embargo, utilizar max_features = 15000 da una similar (incluso mayor a veces), por lo que decidimos dejarla en eso

Ingeniería de atributos

- Largo del documento
- Cantidad de palabras
- Cantidad de letras

Largo del documento

IngAtributos-LargoTweet.ipynb
Se calculo el largo del documento y se agrego como una columna más.

```
0.9201121233356693
```

Como se puede ver, aumento un poco el accuracy utilizando el largo del tweet.

Cantidad de palabras

IngAtributos-CantidadPalabras.ipynb
La accuracy aumento, aunque por muy poco:

```
0.920812894183602
```

Cantidad de letras

IngAtributos-CantidadDeLetras.ipynb
Primero probamos con letras: `"abcdefghijklmnñopqrstuvwxyz":`

```
0.9215136650315346
```

La accuracy sigue aumentando poco a poco

Luego también probamos contando cualquier cosa que no sea un espacio, pero el accuracy fue casi el mismo, por lo que se ve que no cambia tanto.

Contar los

Esta notebook quedo mezclada con la de covid
Dado que cuando borramos los # en un experimento anterior la accuracy bajo tanto, realmente confiaba que contar los hashtag iba a beneficiar al algoritmo. Sin embargo, la accuracy ni se movio.

Realmente confiaba en que por el motivo anterior esto fuera a servir. Pense que quizas el problema venia por el hecho de que son valores muy pequeños, e intente darles más peso (Multiplique el valor del resultado -tampoco se si esto tiene sentido pero estoy corto de tiempo y me divertía probar). Aumento el accuracy pero solo un 0.005%.

```
def contarHashtags(row):
  count = 0;
  for letter in row['tweet']:
    if letter == "#":
      count = count +1;
  return count * 10
```

```
train['Cantidad de #'] = train.apply(lambda row : contarHashtags(row),
axis=1)
```

Accuracy:

```
0.9351786965662229
```

Contar los @

Un experimento que se me ocurrio, pero no hizo más que bajar la accuracy

```
def contarArrobas(row):
  count = 0;
  for letter in row['tweet']:
    if letter == "@":
      count = count +1;
  return count
```

```
train['cantidad de @'] = train.apply(lambda row : contarArrobas(row),
axis=1)
```

Accuracy:

```
0.9320252277505255
```

En varias iteraciones bajo. Muy poco (0.01%), pero bajo.

Tiene la palabra Covid

IngAtributos-HashtagsYCovid.ipynb

```
def contarCovid(row):
  count = 0;
  if "covid" in row['tweet']:
    return 1
  return 0
```

```
0.9358794674141556
```

Aumento un poco la accuracy, por lo que lo voy a dejar.

Modelo Deep Learning

Por simpleza, voy a hacer los experimentos en una notebook:
Experimentos-ModeloRedNeuronal.ipynb

Regularizadores

Early Stopping

Se uso un callback de EarlyStopping para ver como influenciaba al modelo. Se hizo un poco más rápido, pero en muchos casos la accuracy bajo luego de utilizarla.

Algunos utilizados:

```
early_stopping = EarlyStopping(monitor='val_loss',
restore_best_weights=True, patience=10)
```

```
early_stopping = EarlyStopping(monitor='val_loss',
restore_best_weights=True, patience=100)
```

```
early_stopping = EarlyStopping(monitor='val_loss',
restore_best_weights=True, patience=200)
```

Quizas más adelante, cuando añada complejidad al modelo, el early stopping se vuelva más incluyente, pero de momento no veo razón de usarlo.

Dropout

```
adam_optimizer='Adam'
loss='binary_crossentropy'
metric='accuracy'

model = Sequential()
model.add(Dense(20, input_shape=(bag_of_words.shape[1],)))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dense(30, use_bias=True))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dense(1, use_bias=True))
model.add(Dropout(0.5))
```

```
model.add(Activation('sigmoid'))
model.compile(loss=loss, optimizer=adam_optimizer, metrics=[metric])
```

Agregar un dropout al final ayudo un poco con el accuracy

Capas de activacion

Al usar la capa ReLU, aumentamos el accuracy.

```
model = Sequential()
model.add(Dense(20, input_shape=(bag_of_words.shape[1],)))
model.add(Activation(LeakyReLU(alpha=0.3)))
model.add(Dense(30, use_bias=True))
model.add(Activation('relu'))
model.add(Dense(1, use_bias=True))
model.add(Dropout(0.5))
model.add(Activation('sigmoid'))
model.compile(loss=loss, optimizer=adam_optimizer, metrics=[metric])

0.9341275402943238
```

Experimento Batch Size

Una cosa que podemos variar es el batch size. El batch es la cantidad de datos que se utiliza para cada paso del descenso por gradiente. Si usamos menos, el descenso por gradiente va a poder ejecutar más veces el algoritmo. Esto no significa que siempre baje el error. Pero si va a subir el tiempo que demora en ejecutar. Lo bajamos de 100 a 16, y ya se nota que el tiempo se triplico: cada batch toma >3 segundos cuando antes tomaba >1 segundo. SIn embargo, la accuracy bajo a pasar de haber disminuido drásticamente el batch size.

Experimentando Epoch size

Empezamos haciendo todos los tests con un epoch de 200. Es un tamaño decente que no demora mucho y da una buena prediccion. Decidimos aumenta el epoch a 1000, pero eso bajo la accuracy.

Fuimos bajando la cantidad poco a poco, hasta que llegamos de nuevo a 200, por lo que dejamos ese valor.

Red Final

RedFinal.ipynb

Ensemble

RedFinal-Ensemble.ipynb
Para terminar, vamos a hacer un ensemble. Voy a generar utilizando la tenica de k-folds, datasets diferentes para entrenar modelos distintos. Luego, la idea es combinarlos con el método ensemble de clasificación “Voto mayoritario”, y compararlos con la anterior. Si mis hipotesis son independientes entre si, por la regla del ensamble la predicción debería ser mayor.

Luego de hacer el ensemble y subirlo a kaggle, vimos que la accuracy era menor que nuestrap reducción anterior. Esto claramente significa que nuestras hipotesis no eran independientes. Con mas tiempo hubieramos visto porque es que son dependientes entre si.