Uninformative_Feature_Extraction

May 7, 2021

1 Uninformative Feature Extraction Notebook

1.0.1 Log Structure

The HDFS log is built the following way: - %d{yy/MM/dd HH:mm:ss}: The date and time - %???: Unknown 2-4 digit code - %p: The priority of the logging event (INFO, WARN, DEBUG, ERROR, etc.) - %c: Category of logging event (class name) - %m: Log message which may or may not contain a block (blk_ID)

1.0.2 Uninformative Features

We will manually extract the following features by parsing the raw log events: - Event datetime - Event anomaly label (used an external file with labeled block IDs) - Event priority - Event category - Words per event - Time interval between events - Events in rolling time window (10min, 1min, 1s) - One hot encoded words in log message

Further feature detection will be done in an unsupervised way using autoencoders.

1.1 Import Libraries

[1]:
```
import pandas as pd
import numpy as np
import regex as re
from datetime import datetime as dt
import itertools
```

1.2 Import Raw Data

This section can be skipped since exported CSV features already exist.

Go directly to the import dataframe section

[2]:
```
## Import Raw Log Data
raw = pd.read_csv('Data/HDFS.log', header=None, sep='\n')[0]

## Import code testing data sets
#raw = raw[0][:20000] # FOR TESTING
#raw = pd.read_csv('Data/HDFS_2k.log', header = None)[0] # FOR TESTING
```

1.3 Code Parsing Section

The following parsing cells should only be run **ONCE**, then all data exported to CSV for quick imports and further data manipulation if necessary.

Since the CSV files exist locally, we skip to the import section for secondary feature extraction.

1.3.1 Get BLock IDs and their Label

[9]:
```
## Import Anomaly Labels
labels = pd.read_csv('Data/anomaly_label.csv')
labels.iloc[:,1][labels.Label == 'Anomaly'] = 1
labels.iloc[:,1][labels.Label == 'Normal'] = 0

length = len(labels)
anomalies = len(labels[labels.Label==1])
print('Len labels: ', length)
print('Anomaly count: ', anomalies)
print('% Anomalous: ', round(anomalies/length*100, 2),'%')
```

```
Len labels:  575061
Anomaly count:  16838
% Anomalous:  2.93 %
```

1.3.2 Parse for Log Event BlockIDs (and label them)

[4]:
```
## Extract Blocks from Log
blocks_in_order = raw.str.findall(r'(blk_.[\d]*)')
unlist_vectorized = np.vectorize(lambda x: x[0])
blocks_in_order = pd.Series(unlist_vectorized(blocks_in_order))


## Label the extracted block ids via conversion dictionary
binarizer = dict(zip(labels.BlockId, labels.Label))
binarizer_vectorized = np.vectorize(lambda x: binarizer[x])
blocks_binarized = pd.Series(binarizer_vectorized(blocks_in_order))

## Create export checkpoint
labeled_blks = pd.DataFrame({'blkID':blocks_in_order, 'anomaly':
 ↪blocks_binarized})
labeled_blks.head()
labeled_blks.to_feather('Data/labeled_blks.feather')
```

1.3.3 Parse for Message Content

[12]:
```
## Extract Messages
full_messages = raw.str.extract(r'((?<=:\s).*)')[0]

## Extract Key Words by Event
```

```
message_keywords = full_messages.str.findall(r'([A-Za-z]+)')

## Extract Features
words_per_event = message_keywords.map(len)
all_words = pd.Series(itertools.chain(*message_keywords))
unique_words = pd.Series(all_words.unique())

## Export to CSV for processing in Databricks / Cloud
message_keywords.to_csv(r'Data/message_keywords.csv', index = False, header =␣
↪False)
words_per_event.to_csv(r'Data/words_per_event.csv', index = False, header =␣
↪False)
all_words.to_csv(r'Data/all_words.csv', index = False, header = False)
unique_words.to_csv(r'Data/unique_words.csv', index = False, header = False)
```

1.3.4 Parse for Event Categories

[ ]:
```
# Extract raw categories
category = raw.str.extract(r'((?<=dfs\.)[^:]*)')[0]
cat_types = sorted(category.unique())
cat_num = len(category)

# Conversion dictionary function
cat_converter = {key:value for (key, value) in zip(cat_types, np.arange(0,␣
↪cat_num))}
cat_converter_vectorized = np.vectorize(lambda x: cat_converter[x])

# Convert
category_numeric = pd.Series(cat_converter_vectorized(category))
```

1.3.5 Parse for Event Priorities

[ ]:
```
# Extract raw priorities
priority = raw.str.extract(r'([A-Z]{2,})')[0]
prio_types = sorted(priority.unique())
prio_num = len(priority)

# Conversion dictionary function
prio_converter = {key:value for (key, value) in zip(prio_types, np.arange(0,␣
↪prio_num))}
prio_converter_vectorized = np.vectorize(lambda x: prio_converter[x])

# Convert
priority_numeric = pd.Series(prio_converter_vectorized(priority))
```

Quick check to ensure all rows were captured and what the category/priority types were

[ ]:
```
# Quick Check
raw_num = len(raw)

print('Categories :', cat_converter,
      '\nNumber of Categories :', len(category.unique()),
      '\n\nPriorities :', prio_converter,
      '\nNumber of Priorities :', len(priority.unique()),
      f'\n\nCheck all {raw_num} rows were captured:', (raw_num == cat_num) ==␣
 ↪(raw_num == prio_num))
```

1.3.6 Parse for DateTime Information

[ ]:
```
# Extract datetimes from Log
dt_series = raw.str[0:13]

# Convert to DateTime bbjects
dt_converter1 = np.vectorize(lambda x: dt.strptime(x,'%y%m%d %H%M%S'))
datetime = pd.Series(dt_converter(dt_series))

# Convert to DateTime Objects if importing from csv?
#dt_converter2 = np.vectorize(lambda x: dt.fromisoformat(x))
#datetime = pd.Series(dt_converter(datetime))
```

1.3.7 Extract time intervals and events inside rolling time window

2) Extract word features. We'll deal with that later as there are a plethora of options for th
3) One hot encode the message keywords.

For now, let's import processed data and create our numeric dataframe..

[39]:
```
# CHECKPOINT HERE
datetime = pd.read_csv('Data/features/datetime.csv', header = None, names =␣
 ↪['datetime'], squeeze = True)


#blocks_binarized = pd.read_csv('Data/features/anomaly_labels.csv', header =␣
 ↪None, names = ['anomaly'], squeeze = True)
#priority_numeric = pd.read_csv('Data/features/priority_numeric.csv', header =␣
 ↪None, names = ['priority'], squeeze = True)
#category_numeric = pd.read_csv('Data/features/category_numeric.csv', header =␣
 ↪None, names = ['category'], squeeze = True)
#words_per_event = pd.read_csv('Data/features/words_per_event.csv', header =␣
 ↪None, names = ['words_per_event'], squeeze = True)
```

[56]:
```
## Create numeric dataframe
log_numeric = pd.DataFrame({'datetime':datetime,
                            'anomaly':blocks_binarized,
                            'priority':priority_numeric,
```

```
                            'category':category_numeric,
                            'words_per_event':words_per_event,
                            # WORD FEATURES
                            })

log_numeric = log_numeric.set_index('datetime')
log_numeric.sample(3)
```

[56]:

datetime	anomaly	priority	category	words_per_event
2008-11-11 06:52:05	0	0.0	7.0	9.0
2008-11-11 07:45:32	0	0.0	5.0	6.0
2008-11-09 20:56:16	0	0.0	7.0	11.0

1.3.8 Compute Time Interval Features

Here we add a column for the time intervals between events so the model has time separation features (ie. how far apart events occur)

We also add a column for the number of events that occur within a 10 minute, 1 minute, and 1 second rolling window.

Numeric Log Checkpoint

[57]:
```
## Get Time Intervals, delta_t
delta_t = datetime-datetime.shift()
delta_t[0] = delta_t[1] - delta_t[1] # set first item to zero
delta_t = np.array(delta_t / np.timedelta64(1, 's'), dtype = int) # convert to␣
 ↪seconds
log_numeric['delta_t'] = delta_t

## Compute Event Counts in Rolling Windos

## Problem: pd.rolling only looks backwards --> we want a forward look ahead␣
 ↪and we don't want diminishing counts near the end of the dataframe if all␣
 ↪events occured within a few seconds.
# Solution Part 1: apply rolling on the reverse series then reverse the answer
# Solution Part 2: add 100,000 extra rows (or more) with a datetime > 600s so␣
 ↪the rolling window includes all events at the starting window

# add temporary rows
rows_added = 100000
cols = log_numeric.columns.tolist()
template_row = pd.DataFrame(np.full((1, len(cols)), 0), columns = cols, index =␣
 ↪[dt(2100,1,1,0,0,0)])
template_row.index.name = 'datetime'
log_numeric = log_numeric.append(template_row.append([template_row]*rows_added))
```

```
# counting
counts_600s = log_numeric.delta_t[::-1].rolling('600s').count()[::-1]
counts_60s = log_numeric.delta_t[::-1].rolling('60s').count()[::-1]
counts_1s = log_numeric.delta_t[::-1].rolling('1s').count()[::-1]

# keep relevant counts
counts_600s = counts_600s[:-(rows_added+1)]
counts_60s = counts_60s[:-(rows_added+1)]
counts_1s = counts_1s[:-(rows_added+1)]

# remove temporary rows
log_numeric = log_numeric.iloc[:-(rows_added+1), :]

# add count features
log_numeric['evnts_in_10min'] = np.array(counts_600s, dtype = int)
log_numeric['evnts_in_1min'] = np.array(counts_60s, dtype = int)
log_numeric['evnts_in_1s'] = np.array(counts_1s, dtype = int)


## Export Numeric Log
log_numeric.to_csv(r'Data/log_numeric.csv', index = True, header = True)
log_numeric.sample(2)
```

[57]:

datetime	anomaly	priority	category	words_per_event	delta_t	evnts_in_10min	evnts_in_1min	evnts_in_1s
2008-11-10 22:57:45	0	0.0	5.0	6.0	0	76425	7442	60
2008-11-09 23:29:34	0	0.0	5.0	5.0	0	72576	6296	64
2008-11-09 20:48:57	0	0.0	4.0	5.0	0	77131	7318	58
2008-11-10 21:02:19	0	0.0	6.0	11.0	0	256940	11983	10
2008-11-11 06:51:56	0	0.0	7.0	9.0	0	274995	111028	375
2008-11-11 09:44:32	0	0.0	7.0	11.0	0	78677	8610	5
2008-11-10 13:56:08	0	0.0	4.0	4.0	0	9201	1658	12
2008-11-10 21:06:18	0	0.0	6.0	11.0	0	109327	30541	146
2008-11-11 06:52:12	0	0.0	7.0	9.0	0	235767	72427	1809
2008-11-11 04:47:28	0	0.0	6.0	11.0	0			

```
2008-11-11 04:47:28          160302          69467          435
```

Numeric Log Checkpoint

[56]:
```
## Import Log numeric if wanting to start here
log_numeric = pd.read_csv('Data/dataframes/log_numeric.csv', index_col =␣
↪'datetime')

## One-Hot encode the dataframe
log_onehot_no_words = pd.get_dummies(log_numeric,␣
↪columns=['priority','category'], prefix=['prio', 'cat'])

## Export to CSV
log_onehot_no_words.to_csv(r'Data/log_onehot_no_words.csv', index = True,␣
↪header = True)

log_onehot_no_words.sample(2)
```

[56]:

datetime	anomaly	words_per_event	delta_t	evnts_in_10min	evnts_in_1min	evnts_in_1s	prio_0	prio_1	cat_0	cat_1	cat_2	cat_3	cat_4	cat_5	cat_6	cat_7	cat_8
2008-11-11 02:30:37	0	9	0	140322	12718	110	1	0	0	0	0	0	0	0	0	1	0
2008-11-10 21:01:27	0	9	0	393924	144204	2719	1	0	0	0							
2008-11-10 11:10:03	0	6	0	78798	7868	49	1	0	0	0							
2008-11-11 02:39:42	0	10	0	76474	7777	7	1	0	0	0							
2008-11-11 07:56:27	0	9	0	229848	63432	2463	1	0	0	0							
2008-11-10 15:00:00	0	6	0	83064	8192	40	1	0	0	0							
2008-11-10 02:03:28	0	5	0	28673	6577	59	1	0	0	0							
2008-11-09 21:02:35	0	5	0	78261	7418	100	1	0	0	0							
2008-11-11 04:43:22	0	9	0	378858	122551	480	1	0	0	0							
2008-11-10 01:15:51	0	5	0	75423	7594	48	1	0	0	0							

2008-11-10 21:01:27	0	0	0	0	0	1	0
2008-11-10 11:10:03	0	0	0	1	0	0	0
2008-11-11 02:39:42	0	0	0	0	0	1	0
2008-11-11 07:56:27	0	0	0	0	0	1	0
2008-11-10 15:00:00	0	0	0	1	0	0	0
2008-11-10 02:03:28	0	0	1	0	0	0	0
2008-11-09 21:02:35	0	0	0	1	0	0	0
2008-11-11 04:43:22	0	0	0	0	0	1	0
2008-11-10 01:15:51	0	0	1	0	0	0	0

One Hot Log (no words) Checkpoint

END OF PARSING ***

Certain columns (priority, category) have more than 2 categorical states (0,1,...,n). To make the categorical data useable for machines, we one-hot encode those variables.

Finally we export our dataframe to CSV.

We will still need to futher process the data based on the model we choose. If we take a time series approach we will need to create a rolling window with matrix information of all variables.

Note: current data did not distinguish anomalies from normal events after a PCA analysis. Lets try looking at word features.

1.4 Tertiary Feature Extraction (Message Keywords)

This was a desperate attempt to extract keywords deemed relevant. The amount of time spent doing this was kept to a minimal so more "computationally clever" code is not exactly present in this memory heavy cell...

[61]:
```
## Import parsed keywords (lists in string form)
#string_message_keywords = pd.read_csv('Data/features/message_keywords.csv',␣
 ↪header = None, names = ['words'], squeeze = True)

## manually entered an exclusion based on 166 unique words found. 111 words (ie.
 ↪ columns) remain
exclusion_set =␣
 ↪{'block','blk','src','dest','BLOCK','mnt','PacketResponder','for','of','size','from','is','

#### old
#str_to_list = np.vectorize(lambda x: np.array(x.strip("[]'").split("', '").
 ↪remove(), object))
#message_keywords = pd.Series(str_to_list(string_message_keywords)) # This line␣
 ↪takes ~ 20 seconds
#one_hot_keywords = message_keywords.str.join('|').str.get_dummies() # This␣
 ↪line takes ~ 30+ min
```

```
#### new
## Convert back to list
## vectorized ((str to list) - excluded keywords) function
str_to_list = np.vectorize(lambda x: set(x.strip("[]'").split("', '")) -␣
 ↪exclusion_set)

## One hot encoding
## THIS LINE CRASHES WINDOWS WHEN RUN ON THE WHOLE SET
#keywords_onehot = pd.Series(str_to_list(string_message_keywords)).str.
 ↪join('|').str.get_dummies()

## Solution: partition data by 2 million rows, stack, remove from memory,␣
 ↪repeat!
## DONT RUN THIS AGAIN UNLESS YOU NEED TO

#part1 = pd.Series(str_to_list(string_message_keywords[0:2000000])).str.
 ↪join('|').str.get_dummies()
#part2 = pd.Series(str_to_list(string_message_keywords[2000000:4000000])).str.
 ↪join('|').str.get_dummies()
#latest_agg = pd.concat([part1, part2], axis=0, join='outer').replace(np.nan,␣
 ↪0).astype(int)
#del part1; del part2

#part3 = pd.Series(str_to_list(string_message_keywords[4000000:6000000])).str.
 ↪join('|').str.get_dummies()
#latest_agg = pd.concat([latest_agg, part3], axis=0, join='outer').replace(np.
 ↪nan, 0).astype(int)
#del part3

#part4 = pd.Series(str_to_list(string_message_keywords[6000000:8000000])).str.
 ↪join('|').str.get_dummies()
#latest_agg = pd.concat([latest_agg, part4], axis=0, join='outer').replace(np.
 ↪nan, 0).astype(int)
#del part4

#part5 = pd.Series(str_to_list(string_message_keywords[8000000:10000000])).str.
 ↪join('|').str.get_dummies()
#latest_agg = pd.concat([latest_agg, part5], axis=0, join='outer').replace(np.
 ↪nan, 0).astype(int)
#del part5

#part6 = pd.Series(str_to_list(string_message_keywords[10000000:])).str.
 ↪join('|').str.get_dummies()
#latest_agg = pd.concat([latest_agg, part6], axis=0, join='outer').replace(np.
 ↪nan, 0).astype(int)
#del part6
```

```
## SAVE
#latest_agg.to_csv(r'Data/keywords_onehot.csv', header = True)
#latest_agg.reset_index().to_feather(r'Data/keywords_onehot.feather')
```

END OF FEATURE EXTRACTION

START OF DATAFRAME JOINING (our two checkpoint dataframes) ***

1.5 Combine Primary Feature Dataframes

1) One hot encoded non-message features

2) One hot encoded message keyword features

A combined dataframe with column names wasn't saved because we won't necessarily need the column names when feeding it into the model.

[6]:
```
## Import the most recent onehot log
log_onehot_no_words = pd.read_csv('Data/pre_combined_dataframes/
 ↪log_onehot_no_words.csv').iloc[:,2:] # dont include datetime or anomaly␣
 ↪column in final output
print(log_onehot_no_words.info())
## Import the keyword onehot dataframe
log_onehot_keywords = pd.read_feather('Data/pre_combined_dataframes/
 ↪keywords_onehot.feather')
print(log_onehot_keywords.info())


## Check shape compatability
if (log_onehot_no_words.shape[0] == log_onehot_keywords.shape[0]):
    ## Check indeces are identical
    if (len(pd.Series(log_onehot_no_words.index == log_onehot_keywords.index).
 ↪unique()) == 1):
        ## COMBINE
        log_onehot_complete = log_onehot_no_words.join(log_onehot_keywords)
        ## Scale
        from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler()
        scaled = scaler.fit_transform(log_onehot_complete); del␣
 ↪log_onehot_complete
        ## Export sparse array to feather / no datetime
        np.save('Data/ONEHOT_SCALED.npy', scaled)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11175629 entries, 0 to 11175628
Data columns (total 16 columns):
 #   Column           Dtype
```

```
---  ------           -----
 0   words_per_event  int64
 1   delta_t          int64
 2   evnts_in_10min   int64
 3   evnts_in_1min    int64
 4   evnts_in_1s      int64
 5   prio_0           int64
 6   prio_1           int64
 7   cat_0            int64
 8   cat_1            int64
 9   cat_2            int64
 10  cat_3            int64
 11  cat_4            int64
 12  cat_5            int64
 13  cat_6            int64
 14  cat_7            int64
 15  cat_8            int64
dtypes: int64(16)
memory usage: 1.3 GB
None
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11175629 entries, 0 to 11175628
Columns: 112 entries, index to route
dtypes: int32(111), int64(1)
memory usage: 4.7 GB
None
```

[12]:
```
print(log_onehot_keywords.info())
sys.getsizeof(log_onehot_keywords)/1000000000, 'GiB'
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11175629 entries, 0 to 11175628
Columns: 112 entries, index to route
dtypes: int32(111), int64(1)
memory usage: 4.7 GB
None
```

[12]:
```
(5.051384452, 'GiB')
```

1.5.1 Sliding Window Matrix Data

Unfortunately, the export for the sliding window dataframe doens't work since the file size exceeds the amount of RAM my computer has. Fortunately, this cell doesn't take a long time to run. As such, we will reuse this code to create the windows in the modelling file where any final pre processing occurs.

[2]:
```
## Load checkpoint
scaled = np.load('Data/ONEHOT_SCALED.npy')
```

```
window_size = 10
windows = []

## Make Windows
windows = [scaled[row:row+window_size] for row in range(scaled.
 ↪shape[0]-window_size)]

print('Object: windows\nSize:', sys.getsizeof(windows)/1000000000, 'GiB')

## Export
# Never works smh...
windows.to_csv('Data/ONEHOT_SCALED_WINDOWS.csv', index = None)
```