documentation

April 1, 2018

0.0.1 IMPORTING BASIC DATA HANDLING LIBRARIES

```
In [573]: import numpy as np

          import pandas as pd
          pd.set_option("display.height",2000)
          pd.set_option("display.max_rows",2000)
          pd.set_option("display.max_columns",2000)
          pd.set_option("display.width",2000)
          pd.set_option("display.max_colwidth",-1)


          import matplotlib
          from matplotlib import pyplot as plt
          matplotlib.style.use("ggplot")

          import seaborn as sns
          sns.set_style("whitegrid")

          import warnings
          warnings.filterwarnings("ignore")
```

```
height has been deprecated.
```

0.0.2 DATA LOAD

```
In [574]: train = pd.read_csv("/home/maniac/Desktop/Kaggle/Computational/train.csv",encoding="IS
          train = pd.DataFrame({"target":train[0],"id":train[1],"date":train[2],"flag":train[3],
```

In the above piece of code, we load the data and provide headers for easier identification of features. The encoding was 8-bit single characters which is used as an option to load the dataset.

0.0.3 DATA CHECK

```
In [575]: train.describe()
```

Out[575]:
```
<div>
<style>
    .dataframe thead tr:only-child th {
        text-align: right;
    }

    .dataframe thead th {
        text-align: left;
    }

    .dataframe tbody tr th {
        vertical-align: top;
    }
</style>
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>id</th>
      <th>target</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>count</th>
      <td>1.600000e+06</td>
      <td>1.600000e+06</td>
    </tr>
    <tr>
      <th>mean</th>
      <td>1.998818e+09</td>
      <td>2.000000e+00</td>
    </tr>
    <tr>
      <th>std</th>
      <td>1.935761e+08</td>
      <td>2.000001e+00</td>
    </tr>
    <tr>
      <th>min</th>
      <td>1.467810e+09</td>
      <td>0.000000e+00</td>
    </tr>
    <tr>
      <th>25%</th>
      <td>1.956916e+09</td>
      <td>0.000000e+00</td>
    </tr>
```

```
    <tr>
      <th>50%</th>
      <td>2.002102e+09</td>
      <td>2.000000e+00</td>
    </tr>
    <tr>
      <th>75%</th>
      <td>2.177059e+09</td>
      <td>4.000000e+00</td>
    </tr>
    <tr>
      <th>max</th>
      <td>2.329206e+09</td>
      <td>4.000000e+00</td>
    </tr>
  </tbody>
</table>
</div>
```

```
In [576]: train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1600000 entries, 0 to 1599999
Data columns (total 6 columns):
text      1600000 non-null object
id        1600000 non-null int64
date      1600000 non-null object
flag      1600000 non-null object
user      1600000 non-null object
target    1600000 non-null int64
dtypes: int64(2), object(4)
memory usage: 73.2+ MB
```

```
In [577]: train.head()
```

```
Out[577]: <div>
<style>
    .dataframe thead tr:only-child th {
        text-align: right;
    }

    .dataframe thead th {
        text-align: left;
    }

    .dataframe tbody tr th {
        vertical-align: top;
    }
```

```
</style>
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>text</th>
      <th>id</th>
      <th>date</th>
      <th>flag</th>
      <th>user</th>
      <th>target</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>@switchfoot http://twitpic.com/2y1zl - Awww, that's a bummer.  You shoulda g
      <td>1467810369</td>
      <td>Mon Apr 06 22:19:45 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>_TheSpecialOne_</td>
      <td>0</td>
    </tr>
    <tr>
      <th>1</th>
      <td>is upset that he can't update his Facebook by texting it... and might cry as
      <td>1467810672</td>
      <td>Mon Apr 06 22:19:49 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>scotthamilton</td>
      <td>0</td>
    </tr>
    <tr>
      <th>2</th>
      <td>@Kenichan I dived many times for the ball. Managed to save 50%  The rest go
      <td>1467810917</td>
      <td>Mon Apr 06 22:19:53 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>mattycus</td>
      <td>0</td>
    </tr>
    <tr>
      <th>3</th>
      <td>my whole body feels itchy and like its on fire</td>
      <td>1467811184</td>
      <td>Mon Apr 06 22:19:57 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>ElleCTF</td>
```

```
        <td>0</td>
      </tr>
      <tr>
        <th>4</th>
        <td>@nationwideclass no, it's not behaving at all. i'm mad. why am i here? becau
        <td>1467811193</td>
        <td>Mon Apr 06 22:19:57 PDT 2009</td>
        <td>NO_QUERY</td>
        <td>Karoli</td>
        <td>0</td>
      </tr>
    </tbody>
  </table>
  </div>
```

```
In [578]: train["target"].unique()
```

```
Out[578]: array([0, 4])
```

```
In [579]: train["flag"].unique()
```

```
Out[579]: array(['NO_QUERY'], dtype=object)
```

```
In [580]: train.shape
```

```
Out[580]: (1600000, 6)
```

Since, the dataset size is 1.6 million X 6, it is efficient to reduce the dataset size and also shuffle it and take a fraction of shuffled dataset as our training set, for easy observations since large datasets tend to take time while training and as the data consists of text analysis, the feature extraction tend to be cumbersome and it generally increases in size.

0.0.4 SHUFFLING AND SAMPLING OF DATASET

0.0.5 IMPORTING *SHUFFLE* LIBRARY

```
In [581]: from sklearn.utils import shuffle
```

```
In [582]: train = shuffle(train)
          train = train.sample(frac=0.2).reset_index(drop=True)
          train = train[:50]
          test = train[50:100]
```

We shuffled the dataset and took 20% of the dataset and then actually only took the 1st 50 samples from the 20% of the dataset.

0.0.6 DATA CHECK

```
In [583]: train.shape

Out[583]: (50, 6)

In [584]: train.head()

Out[584]: <div>
          <style>
              .dataframe thead tr:only-child th {
                  text-align: right;
              }

              .dataframe thead th {
                  text-align: left;
              }

              .dataframe tbody tr th {
                  vertical-align: top;
              }
          </style>
          <table border="1" class="dataframe">
            <thead>
              <tr style="text-align: right;">
                <th></th>
                <th>text</th>
                <th>id</th>
                <th>date</th>
                <th>flag</th>
                <th>user</th>
                <th>target</th>
              </tr>
            </thead>
            <tbody>
              <tr>
                <th>0</th>
                <td>@TomFelton ohmyGod, you're so hot. i love you! i'm a big fan. can't wait to
                <td>2002111740</td>
                <td>Tue Jun 02 03:10:52 PDT 2009</td>
                <td>NO_QUERY</td>
                <td>helloolivia</td>
                <td>4</td>
              </tr>
              <tr>
                <th>1</th>
                <td>I am sitting at work looking outy at the rain</td>
                <td>2204755845</td>
                <td>Wed Jun 17 03:02:51 PDT 2009</td>
```

```
      <td>NO_QUERY</td>
      <td>sbrady3340</td>
      <td>0</td>
    </tr>
    <tr>
      <th>2</th>
      <td>I'm here mami.  @BitterSweetzz: I miss my friends</td>
      <td>2260873629</td>
      <td>Sat Jun 20 19:54:34 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>Maurayne</td>
      <td>0</td>
    </tr>
    <tr>
      <th>3</th>
      <td>heads stuck in my math textbook...not a good day</td>
      <td>1994987880</td>
      <td>Mon Jun 01 12:46:24 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>NichaelKelly</td>
      <td>0</td>
    </tr>
    <tr>
      <th>4</th>
      <td>Fly with me</td>
      <td>2071846697</td>
      <td>Sun Jun 07 20:00:46 PDT 2009</td>
      <td>NO_QUERY</td>
      <td>heykyeh</td>
      <td>4</td>
    </tr>
  </tbody>
</table>
</div>
```

Now, we need to process our dataset for feeding into the algorithm. The preprocessing can be divided into 8 steps : 1) We replace '4' which is a positive sentiment with '1' for better understanding as a binary classification, where '0' tends to be negative sentiment. 2) We process our dataset using "BeautifulSoup" in order to remove all HTML tags, in case there are any as the dataset is fetched from internet. 3) Using regular expression for treating the text. 4) Stemming the text. 5) Lemmatizing the text. 6) Using "CountVectorizer" to create tokens using the available vocabulary and by using the stopwords. 7) Removal of punctuation marks. 8) Returning the cleaned text into a list.

0.0.7 REPLACING THE VALUE OF *"4"* WITH *"1"* & DROPPING USELESS FEATURES EXCEPT THE *"TEXT"* AND *"TARGET"*

```
In [585]: train["target"] = train["target"].replace(4,1)
          train.drop(["id","date","flag","user"],axis=1,inplace=True)
```

0.0.8 CHECKING DATA

```
In [586]: train.shape

Out[586]: (50, 2)

In [587]: train["target"].unique()

Out[587]: array([1, 0])
```

0.0.9 IMPORTING THE LIBRARIES FOR DATA PREPROCESSING

```
In [588]: from bs4 import BeautifulSoup
          import re
          from nltk.corpus import stopwords
          from nltk.stem.wordnet import WordNetLemmatizer
          from nltk.stem.porter import PorterStemmer
          import string
          from sklearn.feature_extraction.text import CountVectorizer
```

0.0.10 CREATING OBJECTS FOR *STEMMER* AND *LEMMATIZER*

```
In [589]: lmtzr = WordNetLemmatizer()
          stemmer = PorterStemmer()
```

0.0.11 CREATING *COUNTVECTORIZER* OBJECT FOR CREATING TOKENS USING THE VOCABULARY AND *STOPWORDS*

```
In [590]: cv = CountVectorizer(analyzer="word",stop_words=stopwords.words("english"),max_feature
```

When creating the countvectorizer object we defined "max_features", which is actually the number of words to be considered into the vocabulary. More features imply more training time but better results.

0.0.12 *"CLEANING"* FUNCTION WHICH INCLUDES ALL THE PREPROCESSING STEPS AS DEFINED ABOVE

```
In [591]: def cleaning(text) :
              text = BeautifulSoup(text).get_text()
              text = "".join([item for item in text if item not in string.punctuation])
              text = " ".join(re.sub(r"(@[A-Za-z0-9]+(tweeted:)?)|([^A-Za-z \t])|(http?\S*)|(htt
              text = text.lower().split()
              text = [lmtzr.lemmatize(word) for word in text]
              text = [stemmer.stem(word) for word in text]
              return (" ".join(text))
```

Now, the function does all the preprocessing as defined above. We need to store in a list all the features as returned by the function. Hence, the cleaned feature list will be a list of lists.

```
In [592]: training_clean = []
          for i in range(0,train["text"].size) :
              training_clean.append(cleaning(train["text"][i]))
```

0.0.13 CHECKING THE SIZE OF PREPROCESSED LIST

```
In [593]: len(training_clean)

Out[593]: 50
```

0.0.14 CONVERTING THE PREPROCESSED LIST INTO TOKENS BY FITTING *COUNTVECTORIZER* FOR GETTING THE VOCABULARY AND TRANSFORMING THE LIST

```
In [594]: train_feats = cv.fit_transform(training_clean)
          train_feats = train_feats.toarray()
```

0.0.15 CHECKING DATA

```
In [595]: train_feats.shape

Out[595]: (50, 302)

In [596]: train_feats[0]

Out[596]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
                 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0], dtype=int64)

In [597]: len(train_feats)

Out[597]: 50

In [598]: len(train_feats[0])

Out[598]: 302
```

From checking data we can see that size of our "train_feat" dataset is 50 which is equal to number of samples that we took after shuffling. The list is actually a numpy array of dimension 50 X 328 where 50 is the number of rows as described above. The column length is 328 which is actually the transformed text using the "CountVectorizer" after considering all the vocabulary in the available 50 samples in the training dataset.

0.0.16 NOW WE NEED TO DESIGN OUR ALGORITHM FOR CLASSIFICATION TASKS AND PREDICTING ON NEW DATA

The design process is as follows: We split the shuffled dataset into training and testing dataset for fitting the algorithm i.e making a decision boundary and then predicting the outcome using the fitted algorithm, respectively. The fitting process actually calculates the parameters of the cost function so that data is separable by a decision boundary.

Before we split the dataset we convert our list of features into a dataframe and add "target" column for splitting.

0.0.17 CONVERTING THE LIST INTO A DATAFRAME AND APPENDING THE *"TARGET"* FROM OUR ORIGINAL DATASET

```
In [599]: train_feats = pd.DataFrame(train_feats)
          train_feats.shape

Out[599]: (50, 302)

In [600]: train_feats.columns

Out[600]: RangeIndex(start=0, stop=302, step=1)
```

0.0.18 IMPORTING LIBRARIES FOR FORMING A TRAINING AND TESTING DATA

```
In [601]: from sklearn.model_selection import train_test_split
          from sklearn.metrics import accuracy_score
```

Now we need to split the dataset into training and testing dataset.

```
In [602]: X_train,X_test,y_train,y_test = train_test_split(train_feats,train["target"],test_size
          y_train = y_train.as_matrix()
          y_test = y_test.as_matrix()
```

0.0.19 CHECKING DATA

```
In [603]: X_train.shape

Out[603]: (40, 302)

In [604]: X_test.shape

Out[604]: (10, 302)

In [605]: y_train.shape
```

```
Out[605]: (40,)

In [606]: y_test.shape

Out[606]: (10,)
```

0.0.20 DESIGNING ALGORITHM

The algorithm design phase is divided as : 1) We define a function for fitting the algorithm on train set. 2) We define a function for predicting the ouput on test set. 3) We design the algorithm from scratch which include the cost function, hypothesis and minimizing the cost function using gradient descent. 4) We check the accuracy score

```
In [607]: def sigmoid(op) :
              res =  1/(1+np.exp(-op))
              return res
```

The above is the hypothesis function which is called the logistic function or the sigmoid function, and hence the name logistic regression. The output lies between 0 and 1 and as a result we predict classes according to the values obtained by the logistic function. If the output lies above 0.5 we predict that it lies in the "1" else "0".

```
In [608]: def weightinit(numfeats) :
              w = np.zeros((1,numfeats))
              intercept = 0
              return w,intercept
```

The above function initiliazes the parameters as a row vector of 0s and "alpha" as the learning rate also as 0.

```
In [609]: def optimize(w,intercept,X,y) :
              m = X.shape[0]
              res = sigmoid(np.dot(w,X.T)+intercept)
              y_t = y.T
              cost = (-1/m)*(np.sum((y_t*np.log(res))+((1-y_t)*(np.log(1-res)))))
              dw = (1/m)*(np.dot(X.T,(res-(y.T)).T))
              dint = (1/m)*(np.sum(res-(y.T)))
              grad = {"dw":dw,"dint":dint}
              return grad,cost
```

In the above function to optimize our algorithm, we calculate the hypothesis by passing the equation of a line and calculate the gradient using gradient descent for both the weights and the intercept and finally return gradient and the cost function.

```
In [610]: def fit(w,intercept,X,y,alpha,iterations) :
              costs = []
              for i in range(iterations) :
                  grads,cost = optimize(w,intercept,X,y)
                  dw = grads["dw"]
```

```
                dint = grads["dint"]
                w = w - (alpha*(dw.T))
                intercept = intercept - (alpha*dint)
                costs.append(cost)

            coeff = {"w":w,"intercept":intercept}
            gradient = {"dw":dw,"dint":dint}
            return coeff,gradient,costs
```

In the above function we predict the final coefficients, gradient and the cost function after every iteration which is returned in the list of "costs".

```
In [611]: def predict(final,m) :
              y_pred = np.zeros((1,m))
              for i in range(final.shape[1]) :
                  if final[0][i] >= 0.5 :
                      y_pred[0][i] = 1
              return y_pred
```

In the above function we predict the final output as the class in which the input falls based upon the output from the hypothesis or the logistic function.

0.0.21 TESTING THE ALGORITHM

```
In [612]: numfeats = X_train.shape[1]
          w,intercept = weightinit(numfeats)
          coeff,gradient,costs = fit(w,intercept,X_train,y_train,alpha=0.0001,iterations=10000)
          w = coeff["w"]
          intercept = coeff["intercept"]
          final_train = sigmoid(np.dot(w,X_train.T)+intercept)
          final_test = sigmoid(np.dot(w,X_test.T)+intercept)
          mtrain = X_train.shape[0]
          mtest = X_test.shape[0]
          yfinaltrain = predict(final_train,mtrain)
          yfinaltest = predict(final_test,mtest)
```

0.0.22 CHECKING ACCURACY

```
In [613]: yfinaltrain[0]

Out[613]: array([ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
                  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [614]: y_train

Out[614]: array([1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1])

In [615]: accuracy_score(yfinaltrain[0],y_train)
```

```
Out[615]: 0.65000000000000002

In [616]: accuracy_score(yfinaltest[0],y_test)

Out[616]: 0.40000000000000002
```