Big Data ILS - Z 604: Final Project

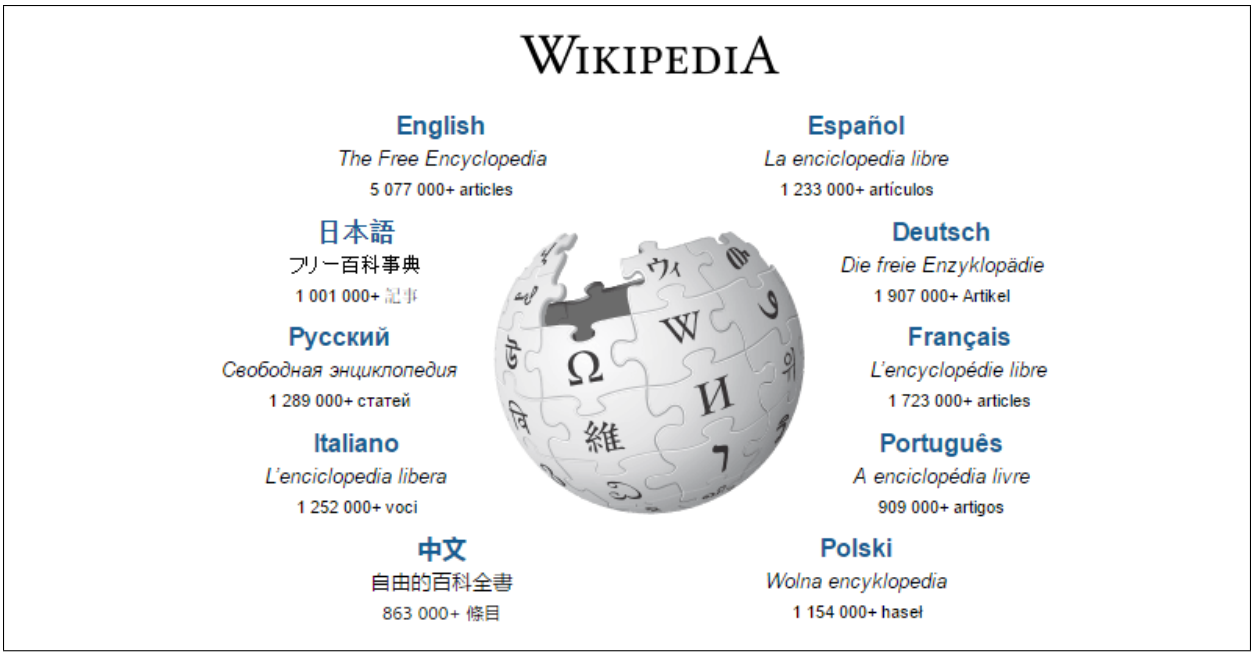
Due on Tuesday, May 3, 2016

Professor Liu

asampath/jashjhav/nramanan/vpatani/sumbhand

Contents

About the Problem



The Question Answered here is a novel one. We are trying to predict the next edit of a certain page. Additionally we also predict whether if a page has their next edit within a certain time period or not. Interestingly enough. You wonder how could this serve as a business solution? Well, this definitely useful to Wikipedia to improve maintainability of their pages.

How? They could assign more resources to a page having more edits and lesser to those which are infrequently edit. This helps them decide what to prioritize and how to make Wikipedia more and more efficient.

Dependencies

We used an array of technologies during the course in order to achieve a superior model.

- Python
- R Programming Language
- MongoDB

Data Formats:

- XML
- CSV
- Pickle

Files

Raw Data

The original data set used for this problem statement is obtained from the **Wikimedia** dump service. We are using the revision data set dumped on 05-03-2016. The following attributes are available in the XML data set -

1. All metadata and data relating to wiki page is inside a page tag.
2. Each page is referred to using a unique id tag.
3. All metadata and data relating to revisions of a page are inside a revision tag. A page can contain multiple revision tags.
4. Each revision has the following tags -
    - An **id** tag to uniquely reference each revision
    - A **timestamp** tag to store the time and date of the revision
    - A **contributor** tag which stores details regarding the user who made the revision. It can contain the username or the IP address of the contributor. This tag may or may not be present.
    - A **text** tag stores the whole text of that page.
    - A **comment** tag which shows any comment for this wiki page.

```
<page>
    <title>Anarchism</title>
    <id>12</id>
    <revision>
      <id>18201</id>
      <timestamp>2002-02-25T15:00:22Z</timestamp>
      <contributor>
        <ip>Conversion script</ip>
      </contributor>
      <minor />
      <comment>Automated conversion</comment>
      <text xml:space="preserve">''Anarchism'' is the political
        theory that advocates the abolition of all forms of
        government.
        ...
      </text>
    </revision>
    <revision>
      <id>19746</id>
      <timestamp>2002-02-25T15:43:11Z</timestamp>
      <contributor>
        <ip>140.232.153.45</ip>
      </contributor>
      <comment>*</comment>
      <text xml:space="preserve">''Anarchism'' is the political
        theory that advocates the abolition of all forms of government.
        ...
```

Indexing

We had begun Indexing the Wiki XML file with Mongo at the beginning, tried transforming it but failed at that attempt. The big problem here was primarily the data size and the unknown data structure. We could not open the file due to its extremely large size and hence we were unsure of the structure.
We had design our own indexing strategy. We are approaching the problem in the following steps:

- First we broke the XML file provided by Wikimedia into chunks so that we can make the file readable.
- This was a major breakthrough since we finally realised the structure of the data.
- We in our script are reading each line and looking for specific tags (Page ID & Revision ID).
- Our indexing Keys are a unique combination Page ID + Revision ID.

```
if "<revision>" in each_line:
    count = count + 1

if "<timestamp>" in each_line:
    datedata = each_line.rstrip().replace('>',' ').replace('<',' ').split()
    temp = datedata[1].replace('T',' ').split()
    date_rev = date_rev + temp[0] + "|"
    #print(temp[0])

if "<id>" in each_line and og_count==0:
    og_count = 1
    each_id = each_line.rstrip().replace('>',' ').replace('<',' ').split()
    page_id = int(each_id[1])
    #print(int(each_id[1]))
```

Feature Construction

Since the Xml data that we got is extremely large, to process the data we break this large XML into small chunks of XML. For every page in the XML, we select certain features and write our feature for every page. Page id is the main index of our generated CSV. Using all of these edit dates for each page we generate more features for prediction of our model. The list of features added to the csv file -

- Page Id
- Number of revisions
- Date for the first revision of a page
- Date for the last revision of a page
- Year-wise count for number of revisions
- Class Label

In approach1 and approach2 discussed below we give a detailed analysis of our newly generated features, the approach used to generate them, the need of these features and their influence on our accuracy.

Model Learning/Training

Our idea is to identify the editing pattern of individual pages in wikipedia. We took two approaches to evaluate our model and to see how it performs based on the provided training data.

Approach I

We try to predict if a particular page would be edited in the next three months. We begin with targeting only those pages with multiple revisions. Timestamps(Date) is a key feature for our model. Every time a page is edited, we have a revision id and date for that revision.We use all of these edit dates in our model. Page id, revision count, previous edit date are some of the significant features we incorporate in this feature. We also generate another feature in this approach, called as a weighted revision factor. The way we do this is - we calculate the no of edits per page in every year since its creation. We give more weights to the edits that occurred post 2012 because the editing pattern is high in the recent years and this factor will greatly enhances our model prediction. Our accuracy increased by around 10-12 % after adding this feature.
For evaluation, we test with by providing a choice of our date and we check for the all the pages if there would be an edit in the next 90 days from the provided date. We do this with a logistic regression and predict for the pages in our test data if they will be edited in next 90 days.

```
{'accuracy': 0.8691588785046729,
 'auc': 0.9892857142857142,
 'confusion_matrix': Columns:
        target_label    int
        predicted_label int
        count   int

 Rows: 3

 Data:
 +--------------+-----------------+-------+
 | target_label | predicted_label | count |
 +--------------+-----------------+-------+
 |      0       |        1        |   14  |
 |      1       |        1        |   35  |
 |      0       |        0        |   58  |
 +--------------+-----------------+-------+
 [3 rows x 3 columns],
 'f1_score': 0.8333333333333333,
 'log_loss': 0.31111022428443974,
 'precision': 0.7142857142857143,
 'recall': 0.9891588785046729
 'roc_curve': Columns:
        threshold       float
        fpr     float
        tpr     float
        p       int
        n       int

 Rows: 100001
```

Approach II

In this approach, we try to predict the next edit date for a given page. This may sound similar to the previous approach but here we are trying to get the exact date of next date. This was a difficult task as only few pages in the data were frequently edited and predicting the next edit date without much previous edit data was challenging. We then tried to get the next edit month instead of the edit and this was relatively simple as the prediction scope was improved. Like in the previous approach, pageid, revision id revision count and edits are used as features here too. Some more important features for this approach are our last edit, second last edit date and average revision days.

- Last edit date is the date when a page was last edited.

- Second last edit is that one edit before the latest revision. Last but one, in precise.
- Average revision days is a feature that we calculated for this approach. It is a ratio of the difference of days with respect to first and last edit date to the total number of revisions. This feature gives us as an idea about the frequency of the page being edited.

With the available features and generated features we predict the month of our next edit. We keep the second last date as a reference and using the average revision days, next edit month is predicted. When we run this on test data, our predicted month and actual month of next edit are compared in our logistic model to evaluate the accuracy.

```
{'accuracy': 0.8873239436619719,
 'auc': 0.8872549019607845,
 'confusion_matrix': Columns:
        target_label    int
        predicted_label int
        count   int

 Rows: 4

 Data:
 +--------------+-----------------+-------+
 | target_label | predicted_label | count |
 +--------------+-----------------+-------+
 |      0       |        0        |   14  |
 |      1       |        1        |   49  |
 |      0       |        1        |   6   |
 |      1       |        0        |   2   |
 +--------------+-----------------+-------+
 [4 rows x 3 columns],
 'f1_score': 0.9245283018867925,
 'log_loss': 0.4117042484663585,
 'precision': 0.8909090909090909,
 'recall': 0.9607843137254902,
 'roc_curve': Columns:
        threshold       float
        fpr     float
        tpr     float
        p       int
        n       int
```

Additional Approach

Random Forest uses an ensemble of decision trees to learn. An ensemble of classifiers has better classification performance than individual classifiers. Also an ensemble portrays better noise resilience. Here decision trees are built from a sample drawn from the training set. Not to forget these samples are with replacement. This prevents over fitting since it decreases the variance without changing the bias.

```
{'accuracy': 0.9722222222222222,
 'auc': 0.9940381558028617,
 'confusion_matrix': Columns:
        target_label    int
        predicted_label int
        count   int

 Rows: 3

 Data:
 +--------------+-----------------+-------+
 | target_label | predicted_label | count |
 +--------------+-----------------+-------+
 |      1       |        0        |   3   |
 |      1       |        1        |   31  |
 |      0       |        0        |   74  |
 +--------------+-----------------+-------+
 [3 rows x 3 columns],
 'f1_score': 0.9538461538461539,
 'log_loss': 0.21196733748805058,
 'precision': 1.0,
 'recall': 0.9117647058823529,
 'roc_curve': Columns:
        threshold       float
        fpr     float
        tpr     float
        p       int
        n       int

 Rows: 100001
```

Future Edit Learning Algorithm

The algorithm we wrote for learning future editing behaviour proceeds as follows:

- Parameter Selection: The main parameters of the learning algorithm are as follows:
    1. Number of edits in certain time intervals prior to 8/4/2008 from the feature universe at each node

for making an optimal splitting decision.

2. Number of Revisions is the Stopping condition on further node splitting. The lower this value, the more this, it under fits or over fits the data.
3. Weighted sum for edits in consecutive interval.
   - Suppose page was edited ni times in the interval 2015-2016 which gets a weight of wi And $n_i - 1$ times in the interval 2015-2016 which gets a weight of $w_i - 1$, Given $w_i > w_i - 1$ Weighted edits for each page would be

$$\sum_{i=0}^{k} W_i n_i / n_i$$

4. Most recent Edit Date
5. First existence of page
6. Text content difference between the first and the most recent version of it, gave very good performance.

- Splitting of Data:
  - Sample Size: The sample size (less than or equal to the fully available sample size) used to train the model, 600 examples here In this case.
  - Test set: The number of examples to learn using the model, here it is 400 examples.

- Logistic Regression

Logistic Regression

1. $\text{logit}(E[y|x]) = \omega^T x$ — logit link function
2. $p(y|x) = \text{Bernoulli}(\alpha)$ — Bernoulli distribution

Where :

$\text{logit}(x) = ln\frac{x}{1-x}$, $y \in \{0, 1\}$, and $\alpha \in (0, 1)$ is the parameter (mean) of the Bernoulli distribution.

It follows that from the data set $D = \{(x_i, y_i)\}$ for i=1 to n is an i.i.d. sample:

$$E[y|x] = \frac{1}{1+e^{-w^T x}}$$

$$p(y|x) = \left(\frac{1}{1+e^{-w^T x}}\right)^y \left(1 - \frac{1}{1+e^{-w^T x}}\right)^{1-y}$$

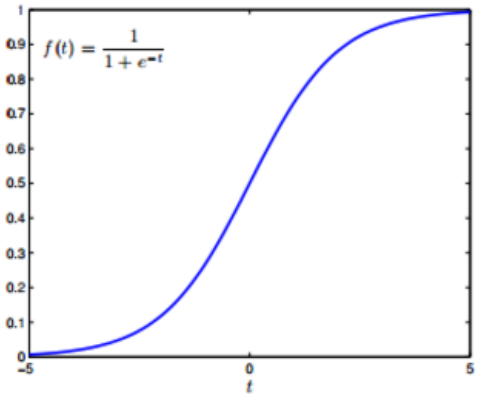
where $\omega = (\omega_0, \omega_1, \ldots, \omega_d)$ is a set of unknown coefficients we want to recover (or learn) from the observed data D.

- We mention the Activation Function

Activation function : Sigmoid function

p(y|x) is a monotonic function of $w^t x$

Below is the Sigmoid function in [-5,5]



Pseudo Code:

```
def sigmoid(self, number):
return 1.0/(1.0+np.exp(number))
```

- Data Conclusion

If P(Y = 1|x, w*) ≥ 0.5 we conclude that data point x should be labeled as positive (ytest = 1).
Otherwise, if P(Y = 1|x, w*) < 0.5, we label the data point as negative (ytest = 0).

Pseudo Code :

```
for i in range(Xtest.shape[0]):
    threshold=self.sigmoid(np.dot(-self.weights.T,Xtest[i]))
    if threshold >= 0.5:
        ytest.append(1)
    else:
        ytest.append(0)
```

- Data Initial Weights

  Initial Weight vectors:

  $w^{(0)} = (X^T X)^{-1} X^T y$ which can be calculated using Ordinary Least Squares regression.

  Determining Weights for each example, stochastically:

  $w^{(t+1)} = w^{(t)} + (X^T P^{(t)} (I - P^{(t)}) X)^{-1} X^T (y - p^{(t)})$

  Term highlighted in green is a partial derivative for every component of w and the above term highlighted in yellow is The Hessian matrix $H_{ij}(w)$ which guarantees that the optimum we find will be a global maximum. p is an n-dimensional column vector of posterior probabilities $p_i = P(Y_i = 1|x_i, w)$, for i = 1, ..., n. P is an n×n diagonal matrix with $P_{ii} = p_i = P(Y_i = 1|x_i, w)$ and I is an n × n identity matrix.

  Note that the second term on the right-hand side of the above equation is calculated in each iteration t; we wrote P and p as $P^{(t)}$ and $p^{(t)}$ to indicate that $w^{(t)}$ was used to calculate them.

  Accuracy calculation:

  We basically normalize over the result, i.e., Countpositive/Countpositive+CountNegative

  Pesudo Code:

```
for i in range(len(ytest)):
        if ytest[i] == predictions[i]:
            correct += 1
    return (correct/float(len(ytest))) * 100.0
```

  Final Model Execution: The final future edits predictor routine then operates on an input as follows: Calculate the prediction for each test set data as a weighted sum of the individual model predictions, using the calculated optimal weight vector.

  Sample Output:

```
C:\Users\Nandini\Documents\Textbooks\Project BD\Classifiers-implemented-master>python script_clas
sify.py
Running on train=400 and test=200 samples
Running learner = Logistic Regression
  sum = sum + (NewWeights[i] - oldweights[i])**2
Accuracy for Logistic Regression: 72.5
```

- We got an accuracy of close to 72.5 which isn't bad.

Conclusions and Interpretation