Final Project
Personality Recognition Based on Text Input

Introduction

In human-computer interaction, the need for adaptive systems in becoming increasingly important. This is especially true when interactive systems fill more meaningful roles. For example, in systems where users interact socially with a virtual agent (chatbot, embodied conversational agent, avatar, etc.), the system needs to be able to both recognize the user to understand the input (text, speech, gestures, etc.) and also generate relevant responses (in the form of text, speech, gestures, etc.).

One approach to understanding the user and generating relevant responses is to recognize the user's personality type. A common personality type classification is the Meyers-Briggs Type Indicator (MBTI), which measures personality on 4 dimensions found in Jungian psychology:

- *extraversion/introversion*, which is known commonly as social energy,
- *sensing/intuition*, which describes how an individuals focuses on information,
- *feeling/thinking*, which describes how people make decisions, and,
- *perceiving/judging*, which describes how people deal with the outside world.

Previous Work

Previously, researchers at Stanford successfully implemented a personality type recognizer model that achieved up to 80% accuracy. However, a shortfall of their work lies in the predictor models utilized. In their work, the authors presented several predictor models, though notable the logistic regression and support vector machine models produced better results. In this work, I show how the use of a more appropriate predictor model can achieve better results with less data pre-processing.

The authors at Stanford performed a lot of data pre-processing. In fact, they (strangely) used both stemming and lemmatization to train the Doc2Vec model for embedding representations of the articles (or documents) in the dataset. They also performed stopword removal, assigned all letters to lowercase versions, and removed all documents with URLs. In total, before pre-processing, there are are 422,844 articles:

- 37463 INTJ samples
- 44456 INTP samples
- 51999 INFJ samples
- 64612 INFP samples
- 6726 ISTJ samples
- 11580 ISTP samples
- 5807 ISFJ samples
- 9029 ISFP samples
- 7774 ENTJ samples
- 24997 ENTP samples
- 6647 ENFJ samples
- 22976 ENFP samples
- 1495 ESTJ samples
- 3328 ESTP samples
- 1489 ESFJ samples
- 1531 ESFP samples

After their pre-processing, the data was reduced by 6% to contain 397,613 samples:

- 35256 INTJ samples
- 41684 INTP samples
- 48860 INFJ samples
- 60329 INFP samples
- 6369 ISTJ samples
- 10794 ISTP samples
- 5446 ISFJ samples
- 8265 ISFP samples
- 7432 ENTJ samples
- 23900 ENTP samples
- 6288 ENFJ samples
- 21956 ENFP samples
- 1461 ESTJ samples
- 3131 ESTP samples
- 1460 ESFJ samples
- 1459 ESFP samples

Approach

Similar to the previous work, my reported models are individual models for each of the personality dichotomies (E/I, S/N, F/T, P/J). Therefore, I trained one model for each binary class (4 models). Each predictor model did use the same Doc2Vec trained model based on the training data input (the text articles). I chose the use of a Doc2Vec model over a Word2Vec model as the model should be making predictions on a document-level rather than word-level basis. Contrary to the baseline model, I did not remove those articles which contained URLs, instead I removed the URL links themselves from the articles. So, in total, there are still 422,844 articles, and I split the data (both the baseline and my pre-processed) into a 75/25 ratio for training and test data, respectively.

Results

My results (average of 10 test runs) are reported in the following tables, for both the baseline (previous work logistic regression) model, as well as my two Naive-Bayes models (Gaussian and Multinomial). Table 1 shows the results when training a Doc2Vec model with $vector\_size = 15$, and Table 2 shows the results when training a Doc2Vec model with $vector\_size = 30$.

I ran several trials of various combinations of parameters for the sklearn models. For both the Gaussian and Multinomial NB models, the $var\_smoothing$ (Gaussian) and $alpha$ (Multinomial) parameters produced better results when they were set to 0.8. The probabilities of the class ($priors$ in Gaussian NB and $class\_priors$ in the Multinomial NB) were found to produce better results (higher accuracy) when the probabilities were not balanced, but in the following ranges:

- E/I, in $[0.3, 0.4]$,
- S/N, in $[0.4, 0.5]$,
- F/T, in $[0.6, 0.7]$,
- P/J. in $[0.5, 0.6]$.

I attribute the results of my model (and the class probabilities) to the disproportionate amount of I/E, S/N, F/T, and P/J articles in the dataset.

Model	**Accuracy (%)**
Baseline	80.1
Gaussian NB	78.7
Multinomial NB	82.3

Table 1: Results: Embedding length=15

Model	**Accuracy (%)**
Baseline	81.6
Gaussian NB	83.6
Multinomial NB	84.4

Table 2: Results: Embedding length=30

Discussion

The results presented could be more improved had I been able to test more data on my machine/environment. In my system, I had to modify the page file sizing to allow for larger allocation for the PyCharm program to build on combined set sizes larger than 1500 (though at the expense of time).

One area for improvement in my solution is to report one F1-score measures to determine just how much my models were really learning about the data. In this way, I could analyze the results for their precision and recall. If the recall was low and the precision was high, then that is decent, but it would be ideal if both measures were high: my model should (ideally) learn more about introversion/extraversion (or the others) when making its predictions. For this, I'd utilize the sklearn.metrics library to report on these results for each of the classes.

Finally, another improvement to this work would be to develop Naive-Bayes predictor models on more-accepted personality indicators (the Big Five model, for example). Another avenue for future work would be to generate the responses that would match the personality type predicted. This would be especially useful in on-the-fly interactive systems for more immersive and human-like representations of virtual agent. For example, in an interactive system with an embodied conversational agent, it would be useful to identify a human interactant by their personality to be more assistive in learning environments, casual interactions, or medical scenarios.

Footnote

The source code of the described program is available at https://github.com/mahdafr/19w_cs5361-labs.

Previous work: http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/dkedia/poster.pdf

```python
import numpy as np
import pandas as pd
import os, math

# fixme: change to dataset location
d = "D:\\Google Drive\\skool\\CS 5361\\datasets\\project\\"
f = "mbti.csv"

class Dataset(object):
    def __init__(self, first_time=False, lo=0.71, hi=0.77, to_load='', chop=0.0005):
        self.X = None; self.Y = None
        self.x = None; self.y = None
        self.lo = lo; self.hi = hi
        self.chop = chop
        self.rand = np.random.randint(1,100000)
        if first_time:
            self._load()    # read from csv (for first run only)
            self._save()    # save the data as npy for faster future runs
        else:
            self._read(to_load)    # read from npy

    """ FIRST RUN ONLY Read the data to be saved for faster runs """
    def _load(self):
        data = np.asarray(pd.read_csv(d + f, sep=',', header=0))   # load from file
        x, y = self._break(data)    # get the data into samples and target values
        self._split(x, y)   # get training and test sets

    """ Read the data to be stored """
    def _read(self, to_load):
        # find the npy files
        lst = os.listdir(d)
        # and ('baseline' not in s) and ('main' not in s) for non-spec use
        file = [i for i, s in enumerate(lst) if (to_load+'npy' in s)]
        # todo randomly load from list of test/train sets
        self.X = np.load(d+lst[file[1]], allow_pickle=True)
        self.Y = np.load(d+lst[file[3]], allow_pickle=True)
        self.x = np.load(d+lst[file[0]], allow_pickle=True)
        self.y = np.load(d+lst[file[2]], allow_pickle=True)
        print('Loaded',self.X.shape[0],'train, ',self.x.shape[0],'test samples.')

    """ Break the data into target values and training samples """
    def _break(self, data):
        X = []; Y = []
        for i in range(data.shape[0]):
            for s in np.array(data[i][1].split('|||')):
                X.append(s)     # sample
                Y.append(data[i][0])    # target value
        return np.asarray(X), np.asarray(Y)

    """ Split the data into training and test sets """
    def _split(self, x, y):
        # randomly choosing a value between lo%-hi% of dataset for train/test split
        breakpoint = np.random.randint(self.lo*x.shape[0], self.hi*x.shape[0])
        self.X = x[:breakpoint]
        self.Y = y[:breakpoint]
        # saving the test data
        self.x = x[breakpoint+1:]
        self.y = y[breakpoint+1:]
        print('Using',self.X.shape[0],'train, ',self.x.shape[0],'test samples.')

    """ Saves the training and test data into npy for efficient access """
    def _save(self, name=''):
        np.save(d+ 'X_' +str(self.X.shape[0]) + name,self.X)
        np.save(d+ 'Y_' +str(self.X.shape[0]) + name,self.Y)
        np.save(d+ 'x_' +str(self.x.shape[0]) + name,self.x)
        np.save(d+ 'y_' +str(self.x.shape[0]) + name,self.y)
        print('Saved npy files to ' +d+name)
```

```python
    """ Getter methods: testing and training data """
    def train(self):
        print('Using',math.ceil(self.chop*self.X.shape[0]),'training samples.')
        self.Y = self._representation(self.Y)
        self.X = self.X[self.rand+math.ceil(self.chop*self.X.shape[0]):]
        self.Y = self.Y[self.rand+math.ceil(self.chop*self.Y.shape[0]):]
        return self.X, self.Y

    def test(self):
        print('Using',math.floor(0.25*self.chop*self.X.shape[0]),'test samples.')
        self.y = self._representation(self.y)
        self.x = self.x[self.rand:self.rand+math.floor(0.25*self.chop*self.x.shape[0])]
        self.y = self.y[self.rand:self.rand+math.floor(0.25*self.chop*self.y.shape[0])]
        return self.x, self.y

    """ Change the representation of target values. """
    def _representation(self, y, one_hot=True):
        if one_hot:
            y = list(s.replace('E', '0') for s in y)
            y = list(s.replace('I', '1') for s in y)
            y = list(s.replace('S', '0') for s in y)
            y = list(s.replace('N', '1') for s in y)
            y = list(s.replace('F', '0') for s in y)
            y = list(s.replace('T', '1') for s in y)
            y = list(s.replace('P', '0') for s in y)
            y = list(s.replace('J', '1') for s in y)
        return np.array(y)

    """ Setter method: testing and training data after pre-processing """
    def set(self, X, Y, x, y, name=''):
        self.X = X
        self.Y = Y
        self.x = x
        self.y = y
        self._save(name)

    """ Get sample count of each personality/class type """
    def of_each(self, in_strings=False):
        if in_strings:
            classes = ['INTJ', 'INTP', 'INFJ', 'INFP', 'ISTJ', 'ISTP', 'ISFJ',
                       'ISFP', 'ENTJ', 'ENTP', 'ENFJ', 'ENFP', 'ESTJ', 'ESTP',
                       'ESFJ', 'ESFP']
        else:
            classes = ['1111', '1110', '1101', '1100', '1011', '1010', '1001',
                       '1000', '0111', '0110', '0101', '0100', '0011', '0010',
                       '0001', '0000']
        for c in classes:
            print(str(np.sum(self.Y == c)), c, 'samples')

    """ Get the target value for c """
    def get_train_target(self, c):
        return np.array([x[c] for x in self.Y])

    def get_test_target(self, c):
        return np.array([x[c] for x in self.y])
```

```python
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
import numpy as np

# fixme: change to dataset location
dr = "D:\\Google Drive\\skool\\CS 5361\\datasets\\project\\"

""" Get the embeddings for each training sample """
def train(d, title, first_time=False):
    X, Y = d.train()
    x, y = d.test()

    if not first_time:
        print('Using ' +title+ ' model')
        model = Doc2Vec.load(dr+title+"d2v.model", )
        return vector(model, X), Y, vector(model, x), y

    train = [TaggedDocument(words=list(X), tags=list(Y))]
    np.save(dr + title + 'train', np.array(train))
    test = [TaggedDocument(words=list(x), tags=list(y))]
    np.save(dr + title + 'test', np.array(test))
    data = [TaggedDocument(words=list(np.append(X,x)),
                           tags=list(np.append(Y,y)))]
    np.save(dr + title + 'data', np.array(data))
    model = _model(title, data, epochs=2)
    return vector(model, X), Y, vector(model, x), y

""" Train the model for a Doc2Vec embedding of input data """
def _model(title, tag, epochs=100, v=15, alpha=0.025):
    # https://medium.com/@mishra.thedeepak/doc2vec-simple
    # -implementation-example-df2afbbfbad5
    model = Doc2Vec(size=v, alpha=alpha, min_alpha=0.00025,
                    min_count=1, dm=1)
    model.build_vocab(tag)

    # print('Training ' + title + 'doc2vec model')
    for epoch in range(epochs):
        model.train(tag,
                    total_examples=model.corpus_count,
                    epochs=model.iter)
        model.alpha -= 0.0002   # decrease the learning rate
        model.min_alpha = model.alpha   # fix to no decay
        # print('Completed epoch', epoch)

    print('Trained ' +title+ 'doc2vec model with epochs=',str(epochs),'vector_size=' + str(v))
    model.save(dr +title+ "d2v_v=" + str(v) + ".model")
    return model

""" Get the feature vector for the classifier """
def vector(model, doc):
    # https://towardsdatascience.com/multi-class-text-classification
    # -with-doc2vec-logistic-regression-9da9947b43f4
    # y, x = zip(*[(d.tags[0],
    #               model.infer_vector(d.words, steps=20))
    #              for d in doc])
    # return x, y
    ret = []
    for i in range(len(doc)):
        ret.append(model.infer_vector(list(doc[i])))
    return np.array(ret)
```

```python
import numpy as np
from nltk import WordNetLemmatizer
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.linear_model import LogisticRegression
from project import dataset, doc2vec as emb

# fixme: change to dataset location
d = "D:\\Google Drive\\skool\\CS 5361\\datasets\\project\\"
title = 'baseline'

""" To preprocess the data """
def second_run(data):
    data.of_each()
    X, Y, x, y = _preprocess(*data.train(), *data.test())
    data.set(X,Y,x,y,name='_'+title)
    data.of_each()

""" One-time use: preprocess the data """
def _preprocess(X,Y,x,y):
    X, Y = _urls_lower_lem_stem_stop(X,Y)
    x, y = _urls_lower_lem_stem_stop(x,y)
    return X, Y, x, y

""" Removes samples with URLs, lemmatizes, stems, then removes stopwords """
def _urls_lower_lem_stem_stop(X, Y):
    nwX = []; nwY = []
    ps = PorterStemmer()
    lem = WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))

    # for each sample
    for i in range(X.shape[0]):
        if 'http' not in X[i]:
            filt = ps.stem(lem.lemmatize(X[i].lower()))
            nwX.append(str([w for w in word_tokenize(filt) if not w in stop_words]))
            nwY.append(Y[i])
    return np.array(nwX), np.array(nwY)

""" Logistic regression """
def build_model(data):
    X, Y, x, y = emb.train(data, title+'_', first_time=True)
    model = LogisticRegression(solver='lbfgs', max_iter=100,multi_class='multinomial')
    print('Classifier:\tLogistic Regression')
    score = []
    for i in range(4):
        score.append(model.fit(X,data.get_train_target(i)).score(x,data.get_test_target(i)))
        print("\tTarget="+str(i),"Score:\t%f" % score[i])
    print("Overall:",str(np.average(score)))

if __name__=="__main__":
    data = dataset.Dataset(to_load=title+'.', chop=0.05)
    # second_run(data)
    build_model(data)
```

```python
import numpy as np
import re
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from project import dataset, doc2vec as emb
from sklearn.preprocessing import MinMaxScaler

# fixme: change to dataset location
d = "D:\\Google Drive\\skool\\CS 5361\\datasets\\project\\"
title = 'main'

""" To preprocess the data """
def second_run(data):
    data.of_each()
    X, Y, x, y = _preprocess(*data.train(), *data.test())
    data.set(X,Y,x,y,name='_main')
    data.of_each()

""" One-time use: preprocess the data """
def _preprocess(X,Y,x,y):
    X, Y = _urls_lower_stem(X,Y)
    x, y = _urls_lower_stem(x,y)
    return X, Y, x, y

""" Removes samples with URLs, and makes samples lowercase """
def _urls_lower_stem(X, Y):
    nwX = []; nwY = []
    ps = PorterStemmer()
    for i in range(X.shape[0]):
        if 'http' in X[i]:  # chop the URL from the string instead
            X[i] = re.sub(
                r"(https?:\/\/)(\s)*(www\.)?(\s)*((\w|\s)+\.)*([\w\-\s]+\/)*([\w\-]+)((\?)?[\w\s]*=\s*[\w\%&]*)*", '', X[i], re.MULTILINE)
        nwX.append(str([w for w in word_tokenize(ps.stem(X[i].lower()))]))
        nwY.append(Y[i])
    return np.array(nwX), np.array(nwY)

""" Naive Bayes' Models """
def build_models(data):
    X, Y, x, y = emb.train(data, title+'_', first_time=True)
    # todo change the parameters for tests
    pred(X, x, data, GaussianNB(), name='Gaussian')
    pred(*rescale(X,x), data, MultinomialNB(class_prior=0.6), name='Multinomial')

""" Train and report on results """
def pred(X, x, data, model, name=''):
    print('Classifier:', name)
    score = []
    for i in range(4):
        score.append(model.fit(X,data.get_train_target(i)).score(x,data.get_test_target(i)))
        print("\tTarget="+str(i),"Score:\t%f" % score[i])
    print("Overall:",str(np.average(score)))

""" Rescales the features for MNB """
def rescale(X, x):
    s = MinMaxScaler()
    s.fit(X)
    return s.transform(X), s.transform(x)

if __name__=="__main__":
    data = dataset.Dataset(to_load=title+'.', chop=0.05)
    # second_run(data)
    build_models(data)
```