UDACITY

PROJECT

Identify Fraud from Enron Email

A part of the Data Analyst Nanodegree Program

PROJECT REVIEW

CODE REVIEW 7

NOTES

poi_id.py 7

```
#!/usr/bin/python
from __future__ import division
from matplotlib import pyplot as plt
from time import time
import numpy as np
import matplotlib
import pandas as pd
import seaborn as sns
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn import svm
from sklearn import neighbors
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import GaussianNB
import collections

from sklearn.preprocessing import StandardScaler, RobustScaler, MinMaxScaler
from sklearn.feature_selection import SelectKBest, f_classif

import sys
import pickle
sys.path.append("../tools/")

from feature_format import featureFormat, targetFeatureSplit
from tester import *

### Task 1: Select what features you'll use.
def select_features(data_dict, filter_pct):
    """ """
    features_list = []
    print "Are there features with many missing values? etc."
    sorted_nan_dict = {}
    for item in data_dict[data_dict.keys()[0]]:
        sorted_nan_dict[item] = sum(x[item]=="NaN" for x in data_dict.values())

    # remove any features that have missing values above the given threshold
    for k,v in sorted(sorted_nan_dict.iteritems(), key=lambda(k,v) : (v,k), \
        reverse = True):
        pct = (v / len(data_dict))
        print " ", k, ": ", v, ",Missing data percentage:", round(pct*100,2), "%"
        # filter out email address which is not a numeric value
        if(pct <= filter_pct and k != "email_address"):
            features_list.append(k)

    #rearrange the list order to have poi in the first position of the list
```

```
    features_list =  [features_list[-1]]+features_list[0:len(features_list)-1:1]

    return features_list


### Task 2: Remove outliers
def remove_outlier(data_dict):
```

AWESOME

Excellent!

```
    data_dict.pop( "TOTAL", 0 )
    data_dict.pop("THE TRAVEL AGENCY IN THE PARK", 0)
    data_dict.pop("LOCKHART EUGENE E", 0)

    return data_dict

### Task 3: Create new feature(s)
def create_new_features(data_dict, features_list):
```

AWESOME

Nice work engineering your new features!

```

    b_ratio_from_poi_fr_msg = False
    b_ratio_to_poi_to_msg = False
    b_total_to_poi_from_poi = False
    b_total_salary_stock = False
    b_total_bonus_stock = False
    for person, item in data_dict.items():

        ## Checking if there's any correlation if the person sends more email
        ## to poi vs all the messages to indicate if he can also be poi
        key = 'ratio_from_poi_fr_msg'
        if not(key in item) and not(key in features_list):
            b_ratio_from_poi_fr_msg = True
            features_list.append(key)

        if b_ratio_from_poi_fr_msg:
            item[key] = computeFraction(item['from_poi_to_this_person'],
                item['from_messages'])

        ## Also checking the other way around
        key = 'ratio_to_poi_to_msg'
        if not(key in item) and not(key in features_list):
            b_ratio_to_poi_to_msg = True
            features_list.append(key)

        if b_ratio_to_poi_to_msg:
            item[key] = computeFraction(item['from_this_person_to_poi'],
                item['to_messages'])

        ## and then check if total combined communication with the poi
        key = 'total_to_poi_from_poi'
        if not(key in item) and not(key in features_list):
            b_total_to_poi_from_poi = True
            features_list.append(key)

        if b_total_to_poi_from_poi:
            item[key] = computeAddition(item['ratio_to_poi_to_msg'],
                item['ratio_from_poi_fr_msg'])

        ## Seeing a high correlation with stock and salary hence using both as
        ## a new total feature
        key = 'total_salary_stock'
        if not(key in item) and not(key in features_list):
            b_total_salary_stock = True
            features_list.append(key)

        if (b_total_salary_stock):
            item[key] = computeAddition(item['salary'],
                item['total_stock_value'])


```

```
        ## Create a new feature which combine both bonus and stock value
        key = 'total_bonus_stock'
        if not(key in item) and not(key in features_list):
            b_total_bonus_stock = True
            features_list.append(key)

        if b_total_bonus_stock:
            item[key] = computeAddition(item['total_stock_value'],item['bonus'])


    return data_dict, features_list



### Task 4: Try a variety of classifiers
def apply_classifiers():
```

AWESOME

Well done attempting so many classifiers

```
    clf = svm.LinearSVC()
    clf2 = neighbors.KNeighborsClassifier(n_neighbors=4, weights="distance", \
        leaf_size=30, algorithm='brute')
    clf3 = GaussianNB()
    clf4 = RandomForestClassifier(n_estimators=100)

    clfs = collections.OrderedDict()
    clfs['LinearSVC'] = clf
    clfs['KNearestNeighbour'] = clf2
    clfs['NaiveBayes'] = clf3
    clfs['RandomForest'] = clf4

    return clfs


### Task 5: Tune your classifier to achieve better than .3 precision and recall
### using our testing script.
def tune_classifier(features_train, labels_train):

    tuned_clfs = collections.OrderedDict()
    svrs = collections.OrderedDict()

    Cs = [0.01, 0.1, 1, 10, 100, 1000, 10000]
    param_grid = {'C': Cs}
    svr = svm.LinearSVC()
    svrs['Tuned LinearSVC'] = [svr, param_grid]

    n_neighbours = range(1,10)
    weights = ['distance', 'uniform']
    algorithms = ['kd_tree', 'brute', 'ball_tree']

    param_grid = {'n_neighbors': n_neighbours, 'algorithm':algorithms, \
                 'weights':weights}
    svr = neighbors.KNeighborsClassifier()
    svrs['Tuned KNearestNeighbour'] = [svr, param_grid]

    n_estimators = range(10,100,10)

    param_grid = {'n_estimators': n_estimators}
    svr = RandomForestClassifier()
    svrs['Tuned RandomForest'] = [svr, param_grid]

    for key, item in svrs.iteritems():
        clf_gs = GridSearchCV(item[0], item[1], scoring='recall_macro')
```

AWESOME

Good optimization of GridCV!

```
        clf_gs = clf_gs.fit(features_train, labels_train)
        clf_gs.best_params_
        print "Best estimator found by grid search:"
        print clf_gs.best_estimator_, "\n"

        tuned_clfs[key] = clf_gs.best_estimator_
```

```

    return tuned_clfs

### Task 6: Dump your classifier, dataset, and features_list so anyone can
### check your results.
def dump_classifier_results(clf, dataset, features_list):
    dump_classifier_and_data(clf, dataset, features_list)


def load_data():

    """ Load the dictionary containing the dataset """
    data_dict = {}
    with open("final_project_dataset.pkl", "r") as data_file:
        data_dict = pickle.load(data_file)

    return data_dict

def plotScatter(data, xlab, ylab, xidx=0, yidx=1):
    """ plot a scatter plot """
    fig = plt.figure()
    ax = fig.add_subplot(111)

    for point in data:
        x = point[xidx]
        y = point[yidx]
        if point[0] == 1.0:
            selcolor = 'r'
        else:
            selcolor = 'b'
        plt.scatter( x, y, color=selcolor, alpha=.4 )

    plt.xlabel(xlab)
    plt.ylabel(ylab)
    plt.show()

def show_data_overview(data_dict, features_list):
    """ show the overall data structure """
    print "\ntotal number of data points: ", len(data_dict)
    print "allocation across classes (POI/non-POI): ", \
        sum(x['poi'] for x in data_dict.values()), "/", \
        sum(x['poi']==0 for x in data_dict.values())
    print "no of features per person: ", len(data_dict[data_dict.keys()[0]])
    print "number of features used: ", len(features_list)
    print "selected features:", ", ".join(features_list)



def computeFraction( poi_messages, all_messages ):
    """ given a number messages to/from POI (numerator)
        and number of all messages to/from a person (denominator),
        return the fraction of messages to/from that person
        that are from/to a POI
   """
    fraction = 0.
    if (poi_messages != "NaN" and all_messages != "NaN"):
        fraction = poi_messages / all_messages

    return fraction

def computeAddition( var1, var2):
    """ does the addition given two list and exclude NaN values """
    total = 0
    if var1 != "NaN" and var2 != "NaN":
        total = var1 + var2

    return total

def plotCorrMatrix(seldata, features_list):
```

AWESOME

Great work with this visualization!

```
    """ plot correlation matrix """
    sns.set(style="white")

    d = pd.DataFrame(data=seldata, columns=features_list)
```

```
    #print d

    # Compute the correlation matrix
    corr = d.corr()

    # Generate a mask for the upper triangle
    mask = np.zeros_like(corr, dtype=np.bool)
    mask[np.triu_indices_from(mask)] = True

    # Set up the matplotlib figure
    f, ax = plt.subplots(figsize=(14, 12))

    # Generate a custom diverging colormap
    cmap = sns.diverging_palette(220, 10, as_cmap=True)

    # Draw the heatmap with the mask and correct aspect ratio
    sns.heatmap(corr, mask=mask, cmap=cmap, vmax=1., center=0, \
            square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=True);

    plt.show()

def split_to_label_features(data_dict, features_list):
    """ Extract features and labels from dataset for local testing """
    data = featureFormat(data_dict, features_list, sort_keys = False)
    labels, features = targetFeatureSplit(data)

    return labels, features

def autoselect_features(data_dict, features_list, sel_k = 5, bShow=True):
    """
    select features based on anova f stats and can take k as no of features
    to be returned.
    """
    labels, features = split_to_label_features(data_dict, features_list)

    # looking for the top k features to use
    selector = SelectKBest(f_classif, k = sel_k)
    selector.fit(features, labels)

    # print the f-stat score sorted in descending order
    if bShow:
        scores = zip(features_list[1:],[round(elem,4) for elem in selector.scores_])
```

AWESOME

Nice work getting the scores!

```
        print 'SelectKBest scores: ', sorted(scores, key=lambda x: x[1], \
            reverse=True)

    final_features = selector.transform(features)

    # Get idxs of columns to keep
    idxs_selected = selector.get_support(indices=True)
    selected_features_list = [features_list[i+1] for i in idxs_selected]

    return final_features, labels, selected_features_list


def apply_robust_feature_scaling(data):
    scaler = RobustScaler()
    scaler.fit(data)
    return scaler.transform(data)

def apply_standard_feature_scaling(data):
    scaler = StandardScaler()
    scaler.fit(data)
    return scaler.transform(data)

def apply_minmax_feature_scaling(data):
    scaler = MinMaxScaler()
    scaler.fit(data)
    return scaler.transform(data)


def clf_score_and_evaluate(cur_clf,features_train, labels_train,features_test, \
```

SUGGESTION

It is great you validate your algorithm, however, there are ways to improve your validation process.

An important part of the validation process is cross-validation, cross-validation generates pairs of train/test sets and these pairs are used to fit/pr many pairs of train/test sets as required in order to generalize results over the entire dataset.

In your validation process implemented it is just generated **one** train/test pair used to fit/predict/evaluate your algorithm. As you can see, this ma

STEPS TO ENHANCE YOUR VALIDATION PROCESS:

1. Create local variables used to store your score values generated in each split, for example:

```
score_all = []
precision_all = []
recall_all = []
```

2. Include in the for loop the fit/prediction/evaluation process and append results to local variables:

```
kf = #Cross-Validation Object
for train_indices, test_indices in kf:
   #make training and testing sets
   features_train= [features[ii] for ii in train_indices]
   features_test= [features[ii] for ii in test_indices]
   labels_train=[labels[ii] for ii in train_indices]
   labels_test=[labels[ii] for ii in test_indices]
   clf = #Algorithm to validate
   clf.fit(features_train,labels_train)
   pred = clf.predict(features_test)
   score_all.append(clf.score(features_test,labels_test))
   precision_all.append(precision_score(labfrom sklearn.preprocessing import MinMaxScalerels_test,pred))
   recall_all.append(recall_score(labels_test,pred))
```

3. Once the for-loop ends, calculate your validation metrics from your local variables:

```
precision = numpy.average(precision_all)
recall = numpy.average(recall_all)
score = numpy.average(score_all)
```

Important: Note this dataset is unbalanced and small, so the splitting technique more appropriated is StratifiedShuffleSplit, since it creates splits complete set. Have a look at this link for further information.

Important 2: Note cross-validation is an iterative process where train/test are generated until results converge, that is, are not dependent on the process is repeated 1000 times!.

```
    labels_test):
    cur_clf.fit(features_train, labels_train)
    pred = cur_clf.predict(features_test)
    sel_avg = 'macro'
    display_precision = 5
    accuracy = round(cur_clf.score(features_test,labels_test),display_precision)
    #print "accuracy: ", round(accuracy,3)

    precision = round(precision_score(labels_test, pred, average=sel_avg), \
        display_precision)
    #print "precision: ", round(precision, 3)

    recall = round(recall_score(labels_test, pred, average=sel_avg), \
        display_precision)
    #print "recall: ", round(recall, 3)

    return accuracy, precision, recall

def compare_algorithms(clfs, features_train, labels_train, features_test, \
    labels_test, sc_features_train, sc_labels_train,sc_features_test, \
    sc_labels_test):

    datamatrix = []
    for name, cur_clf in clfs.iteritems():

        accuracy, precision, recall = clf_score_and_evaluate(cur_clf, \
            features_train, labels_train,features_test, labels_test)

        #print "\nRescaled features:"
        sc_accuracy, sc_precision, sc_recall = clf_score_and_evaluate(cur_clf, \
            sc_features_train, sc_labels_train,sc_features_test, sc_labels_test)

        datarow = [name,accuracy, precision, recall,sc_accuracy, sc_precision,\
        sc_recall]
```

```
        datamatrix.append(datarow)

    return datamatrix

def test_algorithms(clfs, dataset, features_list):
    for name, clf in clfs.iteritems():
        test_classifier(clf, dataset, features_list)

def apply_cross_validation(features, labels, scaled_features):
    features_train, features_test, labels_train, labels_test = \
    train_test_split(features, labels, test_size=0.3, random_state=42)

    sc_features_train, sc_features_test, sc_labels_train, sc_labels_test = \
    train_test_split(scaled_features, labels, test_size=0.3, random_state=42)

    return features_train, features_test, labels_train, labels_test, \
    sc_features_train, sc_features_test, sc_labels_train, sc_labels_test

def get_features_score(features, labels):
    """ use linear svc to compare the features score """
    features_train, features_test, labels_train, labels_test = \
        train_test_split(features, labels, test_size=0.3, random_state=42)

    clf = neighbors.KNeighborsClassifier(n_neighbors=4, weights="distance", \
        leaf_size=30, algorithm='brute')

    accuracy, precision, recall = clf_score_and_evaluate(clf, \
        features_train, labels_train,features_test, labels_test)

    return accuracy, precision, recall
```

▸ readMe.rmd

RETURN TO PATH

Rate this review

Student FAQ