Report - Subtask1

> Research on (MBTI) Myers-Briggs Personality Type Dataset - Ensemble Technique(Random Forest)

Report - Subtask1
- 1 Abstract
- 2 Random Forest
  - 2.1 Logic
  - 2.2 A simple example
  - 2.3 Appendix - Decision Tree
- 3 Preprocessing
- 4 Training
- 5 Separate and overall benchmarks
- 6 Further study
- 7 Conclusion
- 8 References

1 Abstract

This report is mainly about the following 5 sections:

1. An implementation of `Random Forest` in Python.

   Although I've implemented a Decision Tree class as the base estimator, unfortunately, without the cython technique, it perform really slow on dataset that large. So the base estimator used in Random Forest is DecisionTreeClassifier imported from sklearn.

2. Preprocessing of the MBTI dataset.
3. Training 4 classifier on those 4 axes, respectively.
   - **clf_ie**: Introversion (I) – Extroversion (E)
   - **clf_ns**: Intuition (N) – Sensing (S)
   - **clf_tf**: Thinking (T) – Feeling (F)
   - **clf_jp**: Judging (J) – Perceiving (P)
4. Separate and overall benchmarks.
5. Interesting research on Kaggle ForumMessages.

---

2 Random Forest

Code in this section is stored in `RF.py` and `DT.py`.

2.1 Logic

The logic of Random Forest is simple and clear:

1. APIs are defined similarly as sklearn does.
2. Use joblib.Parallel technique to accelerate the fitting process among n_estimators.
3. Prediction is determined by all estimators together, that is, each tree vote for its prediction, and majority wins.

```
from joblib import Parallel, delayed
import pickle as pk
import pandas as pd
import numpy as np
import random as rd
import time
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

N_JOBS = 4


class RandomForest:
    """A RandomForest classifier.

    Parameters
    ----------
    n_estimators : int, optional (default=10)
        The number of estimators in the forest.

    max_depth : int or None, optional (default=None)
        The maximum depth of each tree.
        If None, then nodes are expanded until all leaves are pure
        or until all leaves contain less than `min_leaf_size` samples.

    verbose : int, optional (default=0)
        The level of debugging message in parallel jobs.
        The higher the number is, more detailed messages are printed.

    min_leaf_size : int, optional (default=2)
        The minimum number of samples required to split an internal(non-leaf) node.

    n_features : int, string or None, optional (default=sqrt)
        The number of features to consider when looking for the best split.

            - If int, then consider `n_features` features at each split, randomly.
            - If "auto", then `n_features=sqrt(n_features)`.
            - If "sqrt", then `n_features=sqrt(n_features)`.
            - If "log2", then `n_features=log2(n_features)`.
            - If None, then `n_features=n_features`.

    n_samples : float, optional (default=0.67)
        The proportion of the number of the rows bootstraped during sampling.

    Attributes
```

```
    ----------
    n_estimators : int
        The number of estimators in the forest.

    max_depth : int or None
        The maximum depth of the tree.

    verbose : int
        The level of debugging message in parallel jobs.
        The higher the number is, more detailed messages are printed.

    min_leaf_size : int
        The minimum number of samples required to split an internal(non-leaf) node.

    n_features :
        The number of features to consider when looking for the best split.

    estimators : list of `DecisionTreeClassifier`
        The list of trees.

    Examples
    --------
    >>> from RF import RandomForest

    >>> train_data = ...

    >>> clf = RandomForest()

    >>> clf.fit(train_data)

    >>> test_sample = ...

    >>> print(clf.predict(test_sample))
    [1]

    References
    ----------

    .. [1] sklearn.ensemble forest.py
    .. [2] sklearn.tree tree.py

    Copyright
    ---------
    KarlSzp
    """

    def __init__(self, n_estimators=10, verbose=0,
                 max_depth=None, min_leaf_size=2,
                 n_samples=0.67, n_features="sqrt"):

        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.min_leaf_size = min_leaf_size
        self.n_features = n_features
        self.n_samples = n_samples
        self.verbose = verbose
        self.estimators = []

    def __bootstrap(self, dataset):
        sampled, unsampled = train_test_split(
            dataset, train_size=self.n_samples, shuffle=True, stratify=dataset[:, -1])
```

```
        return sampled, unsampled

    def __buildEstimator(self, sampled):
        clf = DecisionTreeClassifier(
            max_depth=self.max_depth, min_samples_leaf=self.min_leaf_size,
max_features=self.n_features, random_state=None)
        clf.fit(sampled[:, :-1], sampled[:, -1])
        return clf

    def fit(self, dataset):
        bootstraped_datasets = [self.__bootstrap(
            dataset) for i in range(self.n_estimators)]
        self.estimators = Parallel(n_jobs=N_JOBS, verbose=self.verbose,
prefer="threads")(
            delayed(self.__buildEstimator)(i[0]) for i in bootstraped_datasets)
        return

    def __predict(self, est, case):
        return est.predict(case)[0]

    def predict(self, case):
        predictions = [self.__predict(est, case) for est in self.estimators]
        return max(set(predictions), key=predictions.count)


def _predictWithComb(dataset, combination):
    n_estimators, max_depth, min_leaf_size = combination

    clf = RandomForest(n_estimators=n_estimators,
                       max_depth=max_depth, min_leaf_size=min_leaf_size)
    clf.fit(dataset)

    acr = 0.0
    for i in dataset:
        if i[-1] == clf.predict(i[:-1].reshape(1, -1)):
            acr += 1
    return acr / dataset.shape[0], combination


def cross_validation(dataset, para_dict):
    _n_estimators = []
    _max_depth = []
    _min_leaf_size = []
    if "n_estimators" in para_dict.keys():
        _n_estimators = para_dict["n_estimators"]
    if "max_depth" in para_dict.keys():
        _max_depth = para_dict["max_depth"]
    if "min_leaf_size" in para_dict.keys():
        _min_leaf_size = para_dict["min_leaf_size"]

    comb = []
    for i in _n_estimators if len(_n_estimators) else [None]:
        for j in _max_depth if len(_max_depth) else [None]:
            for k in _min_leaf_size if len(_min_leaf_size) else [2]:
                comb.append((i, j, k))

    res = Parallel(n_jobs=N_JOBS, backend="threading")(
        delayed(_predictWithComb)(dataset, i) for i in comb)

    return res[np.argmax([r[0] for r in res])]
```

2.2 A simple example

```
from RF import RandomForest

train_data = ...
test_sample = ...

clf = RandomForest()
clf.fit(train_data)
print(clf.predict(test_sample))
```

2.3 Appendix - Decision Tree

Although I use Decision Tree from sklearn in order to accelerate through Cython technique, I also implement a decision tree, as follow:

```
import pickle as pk
import pandas as pd
import numpy as np
import threading
from joblib import Parallel, delayed
import time
import random as rd
from tqdm import tqdm


class DecisionTree:
    """A decision tree(CART) classifier.

    Parameters
    ----------
    dataset : pandas.DataFrame or list
        The training dataset used to generate the tree.

    max_depth : int or None, optional (default=None)
        The maximum depth of the tree.
        If None, then nodes are expanded until all leaves are pure
        or until all leaves contain less than `min_leaf_size` samples.

    min_leaf_size : int, optional (default=2)
        The minimum number of samples required to split an internal(non-leaf) node.

    n_features : int, string or None, optional (default=sqrt)
        The number of features to consider when looking for the best split.

            - If int, then consider `n_features` features at each split, randomly.
            - If "auto", then `n_features=sqrt(n_features)`.
            - If "sqrt", then `n_features=sqrt(n_features)`.
            - If "log2", then `n_features=log2(n_features)`.
            - If None, then `n_features=n_features`.

    Attributes
    ----------
    max_depth : int or None
        The maximum depth of the tree.

    min_leaf_size : int
        The minimum number of samples required to split an internal(non-leaf) node.
```

```
    dataset : numpy.ndarray
        The training material.

    features : list of string
        Features retrieved from dataset.

    labels : numpy.ndarray
        Labels retrieved from dataset.

    n_features :
        The number of features to consider when looking for the best split.

    root : dict
        The tree root built with __generateDecisionTree().

    Examples
    --------
    >>> from DT import DecisionTree

    >>> train_data = ...

    >>> clf = DecisionTree(dataset=train_data, max_depth=10,
    ...                    min_leaf_size=5, n_features="auto")

    >>> test_sample = ...

    >>> print(clf.predict(test_sample))
    [1]

    References
    ----------

    .. [1] sklearn.ensemble forest.py
    .. [2] sklearn.tree tree.py

    Copyright
    ---------
    KarlSzp
    """

    def __init__(self, dataset, max_depth=None, min_leaf_size=2, n_features="sqrt"):

        self.max_depth = max_depth
        self.min_leaf_size = min_leaf_size

        if isinstance(dataset, pd.DataFrame):
            self.features = dataset.columns[:-1].to_list()
            self.dataset = dataset.values
        elif isinstance(dataset, np.ndarray):
            self.features = list(range(dataset.shape[1]-1))
            self.dataset = dataset
        elif isinstance(dataset, list):
            self.features = dataset[0][:-1]
            self.dataset = np.array([x[:-1] for x in dataset[1:]])
        else:
            raise ValueError("dataset should be a DataFrame or a 2-d list.")

        if isinstance(n_features, int):
            self.n_features = n_features
        elif isinstance(n_features, str):
```

```
            if n_features == "auto" or n_features == "sqrt":
                self.n_features = np.int(np.sqrt(len(self.features)))
            elif n_features == "log2":
                self.n_features = np.int(np.log2((len(self.features))))
            else:
                raise ValueError(
                    "n_features only support methods 'auto', 'sqrt' and 'log2'.")
        elif n_features is None:
            self.n_features = len(self.features)
        else:
            raise ValueError(
                "n_features should be of type int, string or None")

        self.root = self.__generateDecisionTree()

    def __str__(self):
        return "hello"

    def __dataSplit(self, index, value, splitted_dataset):
        left = splitted_dataset[splitted_dataset[:, index] < value]
        right = splitted_dataset[splitted_dataset[:, index] >= value]
        return left, right

    def __gini(self, splitted_dataset):
        labels = [sample[-1] for sample in splitted_dataset]
        labels_counts = [labels.count(label) for label in set(labels)]
        probs = [prob/len(splitted_dataset) for prob in labels_counts]
        return 1 - np.sum(np.power(probs, 2))

    def __giniIndex(self, splitted_datasets):
        gini_index = 0.0
        total_size = np.sum([len(x) for x in splitted_datasets])
        for splitted_dataset in splitted_datasets:
            ratio = len(splitted_dataset) / total_size
            gini_index += ratio * self.__gini(splitted_dataset)
        return gini_index

    def __getSplitPoint(self, splitted_dataset):
        features = rd.sample(
            range(0, len(splitted_dataset[0])-1), self.n_features)

        b_score, b_index, b_value, b_splits = 1, 0, 0, None
        for index in tqdm(features):
            ginis = [(index, self.__giniIndex(self.__dataSplit(index, row[index],
splitted_dataset)), row[index])
                     for row in splitted_dataset]
            min_gini = np.argmin(ginis, axis=0)[1]
            if b_score > ginis[min_gini][1]:
                b_index = ginis[min_gini][0]
                b_score = ginis[min_gini][1]
                b_value = ginis[min_gini][2]
                b_splits = self.__dataSplit(b_index, b_value, splitted_dataset)

        return {'index': b_index,
                'value': b_value,
                'score': b_score,
                'splits': b_splits}

    def __vote(self, splitted_dataset):
        labels = [sample[-1] for sample in splitted_dataset]
        res = max(set(labels), key=labels.count)
```

```python
        return res

    def __split(self, node, depth):
        left, right = node['splits']
        del node['splits']
        if not len(left) or not len(right):
            node['left'] = node['right'] = self.__vote(
                np.append(left, right, axis=0))
            return

        if self.max_depth is not None and depth >= self.max_depth:
            node['left'], node['right'] = self.__vote(left), self.__vote(right)
            return

        if len(left) <= self.min_leaf_size:
            node['left'] = self.__vote(left)
        else:
            node['left'] = self.__getSplitPoint(left)
            self.__split(node['left'], depth + 1)

        if len(right) <= self.min_leaf_size:
            node['right'] = self.__vote(right)
        else:
            node['right'] = self.__getSplitPoint(right)
            self.__split(node['right'], depth + 1)

    def __generateDecisionTree(self):
        print(">>> generating...")
        root = self.__getSplitPoint(self.dataset)
        self.__split(root, 1)
        return root

    def __predict(self, node, case):
        if case[node['index']] < node['value']:
            if isinstance(node['left'], dict):
                return self.__predict(node['left'], case)
            else:
                return node['left']
        else:
            if isinstance(node['right'], dict):
                return self.__predict(node['right'], case)
            else:
                return node['right']

    def predict(self, case):
        return self.__predict(self.root, case)
```

3 Preprocessing

Preprocessing is done in `Preprocessor.ipynb`.

As text material, data is preprocessed and store as `tf-idf`.

This notebook help pre-process the dataset with following steps:

- Separate each post.
- Clean redundant content in posts.
- Separate type into four subtype.

with pickle,

`Preprocessed dataframe is stored as df.pk`

`Preprocessed tf-idf(term-frequency times inverse document-frequency) dataframe is stored as tfidf_df.pk`

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import pickle as pk
from utilities import clean_posts
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer

dataset = "../../Dataset/mbti-type/mbti_1.csv"
```

1 Load and preprocess

In [2]:
```
# Load data & Separate posts
df = pd.read_csv(dataset)
sep_posts = [df['posts'][i].split('|||') for i in range(df.shape[0])]
df = pd.concat([df['type'], pd.Series(sep_posts, name="sep_posts")], axis=1)
df.head()
```

Out[2]:

	type	sep_posts
0	INFJ	['http://www.youtube.com/watch?v=qsXHcwe3krw, ...
1	ENTP	['I'm finding the lack of me in these posts ve...
2	INTP	['Good one _____ https://www.youtube.com/wa...
3	INTJ	['Dear INTP, I enjoyed our conversation the ...
4	ENTJ	['You're fired., That's another silly misconce...

In [3]:
```
df.sep_posts = df.sep_posts.apply(lambda x: ' '.join(x))
df.sep_posts = df.sep_posts.apply(clean_posts)
```

In [4]:
```
df['IE'] = df['type'].apply(lambda x: 1 if x[0] == 'E' else 0)
df['NS'] = df['type'].apply(lambda x: 1 if x[1] == 'S' else 0)
df['TF'] = df['type'].apply(lambda x: 1 if x[2] == 'F' else 0)
df['JP'] = df['type'].apply(lambda x: 1 if x[3] == 'P' else 0)
```

2 Build Vectorizer

Vectorizer is built with:

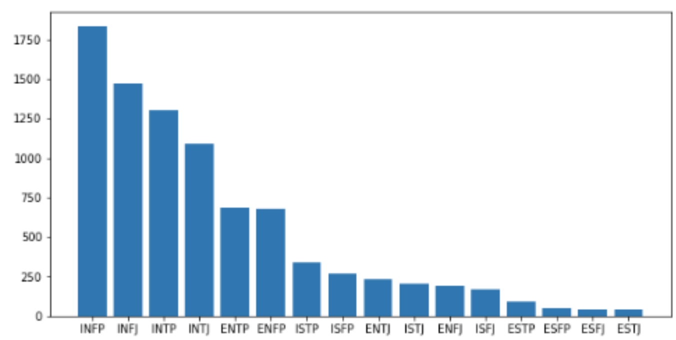
- `CountVectorizer`
- `TfidfTransformer`

in `sklearn.feature_extraction.text`.

In [5]:
```
# Plot and observe the distribution
types = df.type.value_counts()
types_name = list(map(lambda x:(x+'s').lower(), types.index))
types_name += list(map(lambda x:x.lower(), types.index))
stop_words = ['and','the','to','of'] + types_name

plt.figure(figsize=(10,5))
plt.bar(types.index, types.values)
```

Out[5]: `<BarContainer object of 16 artists>`



In [6]:
```python
# Init Vectorizer
Vectorizer = CountVectorizer(stop_words=stop_words,
                             max_features=2000,
                             analyzer="word",
                             max_df=0.8, min_df=0.1)
```

In [7]:
```python
# Build term-document matrix
corpus = df.sep_posts.values.reshape(1,-1).tolist()[0]
td_matrix = Vectorizer.fit_transform(corpus).toarray()
```

In [8]:
```python
# Transform a count matrix to a normalized
# (1) term-frequency or
# (2) term-frequency times inverse document-frequency
# representation.

Transformer = TfidfTransformer()
tfidf_matrix = Transformer.fit_transform(td_matrix).toarray()
tfidf_df = pd.DataFrame(tfidf_matrix, columns=Vectorizer.get_feature_names())
```

In [9]:
```python
tfidf_df_IE = pd.concat([tfidf_df, df['IE']], axis=1)
tfidf_df_NS = pd.concat([tfidf_df, df['NS']], axis=1)
tfidf_df_TF = pd.concat([tfidf_df, df['TF']], axis=1)
tfidf_df_JP = pd.concat([tfidf_df, df['JP']], axis=1)
```

3 Storage

In [10]:
```python
# Store Vectorizer and Transformer
with open('./Vectorizer.pk', 'wb') as pkl:
    pk.dump(Vectorizer, pkl)

with open('./Transformer.pk', 'wb') as pkl:
    pk.dump(Transformer, pkl)
```

In [11]:
```python
# Store dataframes
with open('./tfidf_df_IE.pk', 'wb') as pkl:
    pk.dump(tfidf_df_IE, pkl)

with open('./tfidf_df_NS.pk', 'wb') as pkl:
    pk.dump(tfidf_df_NS, pkl)

with open('./tfidf_df_TF.pk', 'wb') as pkl:
    pk.dump(tfidf_df_TF, pkl)

with open('./tfidf_df_JP.pk', 'wb') as pkl:
    pk.dump(tfidf_df_JP, pkl)
```

In [12]:
```python
# Store dataframe
with open('./df.pk', 'wb') as pkl:
    pk.dump(df, pkl)

with open('./tfidf_df.pk', 'wb') as pkl:
    pk.dump(tfidf_df, pkl)
```

In [13]:
```python
# Store csv
df.to_csv('./sep_mbti.csv', index=False)
```

4 Training

Training and bench marking is done in `mbti-random-forest.ipynb`.

mbti-random-forest

1 Import packages and load preprocessed dataframe

In [15]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import re
import string
import pickle as pk
import time
from tqdm import tqdm
from joblib import Parallel, delayed
from utilities import clean_posts, postVectorizer
from RF import RandomForest, cross_validation
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from sklearn.svm import SVC
from xgboost import XGBClassifier,plot_importance

method_dict = {
    'RF': 'RandomForest',
    'SVM': 'SVM',
    'XGB': 'XGBoost',
    'DL': 'DeepLearning'
}

type_dict = {
    0: ['I', 'N', 'T', 'J'],
    1: ['E', 'S', 'F', 'P']
}

with open('./type_explanation.pk', 'rb') as pkl:
    type_explanation = pk.load(pkl)

type_keys = list(type_explanation.keys())
```

In [2]:
```
with open('./tfidf_df.pk', 'rb') as pkl:
    tfidf_df = pk.load(pkl)

with open('./df.pk', 'rb') as pkl:
    df = pk.load(pkl)

with open('./clf_ie.pk', 'rb') as pkl:
    clf_ie = pk.load(pkl)

with open('./clf_ns.pk', 'rb') as pkl:
    clf_ns = pk.load(pkl)

with open('./clf_tf.pk', 'rb') as pkl:
    clf_tf = pk.load(pkl)

with open('./clf_jp.pk', 'rb') as pkl:
    clf_jp = pk.load(pkl)
```

2 Training and Benchmarking

In [3]:

```
# train_by_type
# @params:
#     _type: (string), type to be classified; types:[IE, NS, TF, JP]
#    method: (string), type of the classifier; methods:['RF', 'SVM', 'XGB']
# benchmark: (boolean), whether benchmark on dataset; benchmark:[True, False]
##

def train_by_type(_type, method='RF', benchmark=False):
    print(">>> Training Type {}".format(_type))
    y = df[_type].values.reshape(-1, 1)
    X_train, X_test, y_train, y_test = train_test_split(tfidf_df.values, y, test_size=0.33, train_size=0.6
                                                        random_state=13, shuffle=True, stratify=y)

    print("# @Training START #")
    if method == 'RF':
        clf = RandomForest(n_estimators=100, verbose=0, min_leaf_size=3)
        clf.fit(np.concatenate((X_train, y_train), axis=1))

    elif method == 'SVM':
        clf = SVC(gamma='auto', probability=True)
        clf.fit(X_train, y_train)

    elif method == 'XGB':
        clf = XGBClassifier()
        clf.fit(X_train, y_train,
                early_stopping_rounds=10,
                eval_metric="logloss",
                eval_set=[(X_test, y_test)],
                verbose=False)
    else:
        raise ValueError("Invalid Method.")

    print("# @Training END #\n")

    time.sleep(0.5)

    if benchmark:
        print("# @Scoring START # --- {:s}".format(method_dict[method]))
        time.sleep(0.5)
        print("Type: {}: {} : {} = {} : {}".format(_type, _type[0], _type[1], sum(y)/len(y),1-sum(y)/len(y
        time.sleep(0.5)
        pred_train = [clf.predict(i.reshape(1,-1)) for i in tqdm(X_train)]
        print("Accuracy on training set - %s" % _type, accuracy_score(y_train, pred_train))
        print("F1 Score on training set - %s" % _type, \
                (f1_score(y_train, pred_train) if len(set(pred_train))>1 else 0.0))
        time.sleep(0.5)
        pred_test = [clf.predict(i.reshape(1,-1)) for i in tqdm(X_test)]
        print("Accuracy on testing set - %s" % _type, accuracy_score(y_test, pred_test))
        print("F1 Score on testing set - %s" % _type, \
                (f1_score(y_test, pred_test) if len(set(pred_test))>1 else 0.0))
        print("# @Scoring END #\n")

    return clf
```

5 Separate and overall benchmarks

For each type, train a type-specified classifier and benchmark individually.

In [4]:
```
# Separately benchmarking
clf_ie = train_by_type('IE', 'RF', benchmark=True)
clf_ns = train_by_type('NS', 'RF', benchmark=True)
clf_tf = train_by_type('TF', 'RF', benchmark=True)
clf_jp = train_by_type('JP', 'RF', benchmark=True)
```

```
>>> Training Type IE
# @Training START #
# @Training END #

# @Scoring START # --- RandomForest
Type: IE: I : E = [0.23043228] : [0.76956772]
100%|██████████| 5812/5812 [00:27<00:00, 208.21it/s]
Accuracy on training set - IE 0.9963867859600826
F1 Score on training set - IE 0.9920963492660896
100%|██████████| 2863/2863 [00:14<00:00, 203.57it/s]
Accuracy on testing set - IE 0.7694725812085226
F1 Score on testing set - IE 0.0
# @Scoring END #

>>> Training Type NS
# @Training START #
# @Training END #

# @Scoring START # --- RandomForest
Type: NS: N : S = [0.13798271] : [0.86201729]
100%|██████████| 5812/5812 [00:27<00:00, 213.57it/s]
Accuracy on training set - NS 0.9850309704060565
F1 Score on training set - NS 0.9426499670402109
100%|██████████| 2863/2863 [00:13<00:00, 217.93it/s]
Accuracy on testing set - NS 0.8620328326929794
F1 Score on testing set - NS 0.0
# @Scoring END #

>>> Training Type TF
# @Training START #
# @Training END #

# @Scoring START # --- RandomForest
Type: TF: T : F = [0.5410951] : [0.4589049]
100%|██████████| 5812/5812 [00:29<00:00, 196.26it/s]
Accuracy on training set - TF 0.9994838265657261
F1 Score on training set - TF 0.9995228248767298
100%|██████████| 2863/2863 [00:14<00:00, 199.33it/s]
Accuracy on testing set - TF 0.7216206776108977
F1 Score on testing set - TF 0.7497645211930927
# @Scoring END #

>>> Training Type JP
# @Training START #
# @Training END #

# @Scoring START # --- RandomForest
Type: JP: J : P = [0.60414986] : [0.39585014]
100%|██████████| 5812/5812 [00:27<00:00, 211.08it/s]
Accuracy on training set - JP 0.9998279421885754
F1 Score on training set - JP 0.9998576107076748
100%|██████████| 2863/2863 [00:13<00:00, 207.13it/s]
Accuracy on testing set - JP 0.6196297589940621
F1 Score on testing set - JP 0.7447855636278415
# @Scoring END #
```

For the whole dataset, combine classifiers above and benchmark.

In [18]:
```python
# predict_full_type
# @params:
#       text: (string), text(post) to predict.
#      _type: (string), True type; types:[INTJ~ESFP]
#    strict: (boolean), if True:
#                           return a tuple of index (predicted, true).
#                       if False:
#                           return a match rate determined by each subtype;
#                           e.g.: INTJ - INTP : 75% matched.
#                       benchmark:[True, False]
##
def predict_full_type(text, _type, strict=True):
    text = postVectorizer(clean_posts(text))
    IE = clf_ie.predict(text)
    NS = clf_ns.predict(text)
    TF = clf_tf.predict(text)
    JP = clf_jp.predict(text)
    match_rate = 0
    pred_type = type_dict[IE][0] + type_dict[NS][1] + type_dict[TF][2] + type_dict[JP][3]

    if strict:
        return type_keys.index(pred_type), type_keys.index(_type)
    else:
        for i in range(4):
            if _type[i] == pred_type[i]:
                match_rate += 25
        print("Predicted Type: {}  |  True Type: {}  |  [{}{}%]Matched".format(
            pred_type, _type, "" if match_rate==100 else " ", match_rate))
        return match_rate
```

In [17]:
```python
# Overall benchmarking strictly

test_size = df.shape[0]

preds = Parallel(n_jobs=4, prefer="threads")\
(delayed(predict_full_type)(df.sep_posts[i], df.type[i], strict=True) for i in tqdm(range(te

print("Overall Accuracy: {}".format(accuracy_score([i[1] for i in preds], [i[0] for i in pre
print("Overall F1 Score: {}".format(f1_score([i[1] for i in preds], [i[0] for i in preds], a
```

```
100%|██████████| 8675/8675 [10:28<00:00, 13.81it/s]
```

```
Overall Accuracy: 0.6893371757925072
Overall F1 Score: 0.5653944459052058
```

In [22]:
```python
# Overall benchmarking non-strictly
test_size = df.shape[0]
acc = Parallel(n_jobs=4, prefer="threads")\
(delayed(predict_full_type)(df.sep_posts[i], df.type[i], strict=False) for i in tqdm(range(t

print("Overall Accuracy: {}".format(np.sum(acc) / (100*test_size)))
```

```
Predicted Type: INTP  |  True Type: INTP  |  [100%]Matched
Predicted Type: ENTP  |  True Type: ENTP  |  [100%]Matched
Predicted Type: INTP  |  True Type: INTJ  |  [ 75%]Matched
Predicted Type: INFP  |  True Type: INTP  |  [ 75%]Matched
Predicted Type: INTJ  |  True Type: INTJ  |  [100%]Matched
```

```
100%|██████████| 8675/8675 [11:03<00:00, 13.07it/s]
```

```
Predicted Type: INTP  |  True Type: ENTP  |  [ 75%]Matched
Predicted Type: ENTP  |  True Type: ENTP  |  [100%]Matched
Predicted Type: INTP  |  True Type: INTJ  |  [ 75%]Matched
Predicted Type: INFP  |  True Type: INFJ  |  [ 75%]Matched
```

```
Predicted Type: ISTP  |  True Type: ISFP  |  [ 75%]Matched
Predicted Type: ENFP  |  True Type: ENFP  |  [100%]Matched
Predicted Type: INTP  |  True Type: INTP  |  [100%]Matched
Predicted Type: INFP  |  True Type: INFP  |  [100%]Matched
Predicted Type: INFP  |  True Type: INFP  |  [100%]Matched
Overall Accuracy: 0.9120172910662824
```

NOTE

1. Those case that f1-score is 0.0 means the prediction incorrectly missed some existed label:

   e.g.: true = [1, 0, 1], prediction = [0, 0, 0] --> f1-score = 0.0

2. In my opinion, the non-strict benchmarker seems to be more dependable.

6 Further study

For further study, I download Kaggle ForumMessages and do an interesting research on it.

3 Further study on Kaggle ForumMessage

In [49]:
```
k_data = pd.read_csv('testdata/ForumMessages.csv').Message.dropna().values
k_texts = []
for i in k_data:
    if len(i) > 1000:
        k_texts.append(i)
```

In [70]:
```
def k_clean(texts):
    texts = [re.sub(r'<code>.*</code>', " ", s) for s in texts]
    texts = [re.sub(r'<[/]?[a-z]+>', "", s) for s in texts]
    return texts

k_texts = k_clean(k_texts)
```

In [93]:
```
res = predict_full_type(k_texts[1], _type=None)
pprint(k_texts[1])
print(res)
pprint(type_explanation[res])
```

```
('hi all i have a question about crossvalidation i am fitting a glm r model to '
 'my training dataset and it gives me an auc numbernumber then i do a '
 'numberfold crossvalidation each of my cross validations comes out with a auc '
 'numbernumber but then when i submit my model it has a aucnumbernumber on '
 'leaderboard what am i doing wrong QST if i am overfitting so badly on '
 'training set i dont understand why doesnt cross validation show that QST '
 'here is my code for crossvalidation data is a dataframe k is number of folds '
 'kfoldglmltfunctiondatak nltasintegernrowdatak errvectltrepnak for i in '
 'numberk snumberltinumbernnumbernumber snumberltin subsetltsnumbersnumber '
 'trainltdatasubset testltdatasubset fit lt glmaction '
 'datatrainfamilyquotbinomialquot prediction lt '
 'predictfitnewdatatesttypequotresponsequot '
 'labelsltasnumericascharactertestnumber err lt rocarealabelspredictiona '
 'errvectilterr returnerrvect cheers anna')
INTP
['1.安静、自持、弹性及具适应力',
 '2.特别喜爱追求理论与科学事理',
 '3.习于以逻辑及分析来解决问题—问题解决者',
 '4.最有兴趣于创意事务及特定工作，对聚会与闲聊无大兴趣',
 '5.追求可发挥个人强烈兴趣的生涯',
 '6.追求发展对有兴趣事务之逻辑解释']
```

7 Conclusion

Thanks for your reading and please refer to (Jupyter notebook necessary):

1. src/mbti-random-forest.ipynb
2. src/RF.py
3. src/DT.py
4. src/utilities.py

for a better experience!

In this experiment, it's clear that:

The model performs bad when making classification on type "I-E" and "N-S".

```
# @Scoring START # --- RandomForest
Type: IE: I : E = [0.23043228] : [0.76956772]
100%|██████████| 6940/6940 [00:32<00:00, 216.31it/s]
Accuracy on training set - IE 0.9463976945244956
F1 Score on training set - IE 0.8683651804670912
100%|██████████| 1735/1735 [00:07<00:00, 218.12it/s]
Accuracy on testing set - IE 0.7694524495677233
F1 Score on testing set - IE 0.0
# @Scoring END #
```

```
# @Scoring START # --- RandomForest
Type: NS: N : S = [0.13798271] : [0.86201729]
100%|██████████| 6940/6940 [00:31<00:00, 216.95it/s]
Accuracy on training set - NS 0.8829971181556195
F1 Score on training set - NS 0.26449275362318836
100%|██████████| 1735/1735 [00:07<00:00, 218.15it/s]
Accuracy on testing set - NS 0.8622478386167147
F1 Score on testing set - NS 0.0
# @Scoring END #
```

While, it performs well on type "T-F" and "J-P".

```
# @Scoring START # --- RandomForest
Type: TF: T : F = [0.5410951] : [0.4589049]
100%|██████████| 6940/6940 [00:33<00:00, 206.11it/s]
Accuracy on training set - TF 0.9998559077809799
F1 Score on training set - TF 0.9998668264748969
100%|██████████| 1735/1735 [00:08<00:00, 216.77it/s]
Accuracy on testing set - TF 0.7066282420749279
F1 Score on testing set - TF 0.7358588479501815
# @Scoring END #
```

```
# @Scoring START # --- RandomForest
Type: JP: J : P = [0.60414986] : [0.39585014]
100%|██████████| 6940/6940 [00:33<00:00, 205.58it/s]
Accuracy on training set - JP 0.9997118155619596
F1 Score on training set - JP 0.9997615641392466
100%|██████████| 1735/1735 [00:08<00:00, 198.48it/s]
Accuracy on testing set - JP 0.6207492795389049
F1 Score on testing set - JP 0.7453560371517027
# @Scoring END #
```

To tell why, I observe the log many times and finally found a possible reason:

The inbalance in training data cause that.

8 References

1. Kaggle - MBTI dataset
2. Myersbriggs - mbti-basics
3. Devdocs - scikit-learn documentation
4. Joblib.Parallel

2020/6

Karl

BACK TO TOP