Report - Subtask2

> Research on (MBTI) Myers-Briggs Personality Type Dataset - SVM(LinearSVC)

Report - Subtask2

- 1 Abstract
- 2 LinearSVC
  - 2.1 LinearSVC() vs SVC(kernel='linear')
  - 2.2 Parameters
- 3 Preprocessing
- 4 Training
- 5 Separate and overall benchmarks
- 6 Further study
  - 6.1 Introduction
  - 6.3 Testcases
- 7 Conclusion
- 8 References

1 Abstract

This report is mainly about the following 5 sections:

1. Usage of LinearSVC in sklearn.svm
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 LinearSVC

LinearSVC is imported from sklearn.svm.

2.1 LinearSVC() vs SVC(kernel='linear')

As a 2-class classification task, linear svm is more cost-effective than those non-linear technique.

In `sklearn`, LinearSVC() and SVC(kernel='linear')(SVC with linear kernel) are both reliable enough, however, LinearSVC() is implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

Consequently, training process in this task use LinearSVC() instead of SVC(kernel='linear').

2.2 Parameters

Parameters in this model is printed below:

In [65]:
```
clf_ie
```

Out[65]:
```
LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=None, tol=1e-05,
          verbose=0)
```

Most of them are set default, some crucial ones are list below:

- C: Penalty parameter C of the error term.
- dual: Select the algorithm to either solve the dual or primal optimization problem.
- penalty: Specifies the norm used in the penalization. The ‘l2’ penalty is the standard used in SVC.
- tol: Tolerance for stopping criteria.

When trying `loss='hinge'` instead of `loss='squared_hinge'`, the classifier never converge:

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

```
/home/karl/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

```
# @Training END #
```

When trying to change other parameters, the result doesn't improve a lot:

In [82]:
```
benchmark_df
```

Out[82]:

	IE	NS	TF	JP	Full Type(Strict)	Full Type(Loose)
Scale	23% : 77%	14% : 86%	54% : 46%	60% : 40%	NaN	NaN
ACC	0.823055	0.881988	0.82147	0.738329	0.426628	0.804006
F1	0.49465	0.321458	0.836823	0.796321	0.230088	NaN

In [54]:
```
benchmark_df
```

Out[54]:

	IE	NS	TF	JP	Full Type(Strict)	Full Type(Loose)
Scale	23% : 77%	14% : 86%	54% : 46%	60% : 40%	NaN	NaN
ACC	0.822334	0.882997	0.824352	0.734294	0.421787	0.803833
F1	0.492384	0.307167	0.839033	0.792856	0.226198	NaN

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

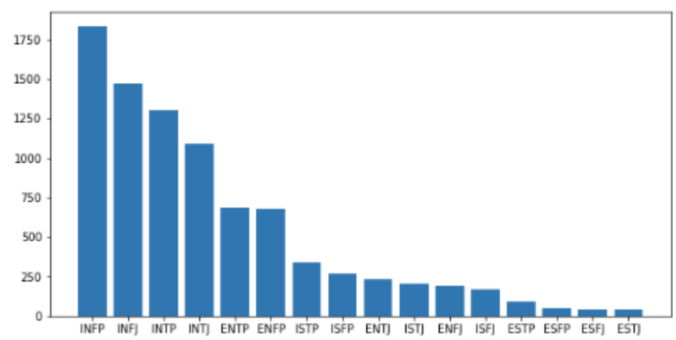
Vectorizer is built with:

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

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

In [8]:
```
# 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]:
```
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]:
```
# 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]:
```
# 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]:
```
# 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]:
```
# Store csv
df.to_csv('./sep_mbti.csv', index=False)
```

4 Training

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

mbti-SVM

1 Import packages and load preprocessed dataframe

In [39]:
```
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 pprint import pprint
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, LinearSVC
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('pickles/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 [42]:
```
benchmark_df = pd.DataFrame(np.zeros((3, 6)), index=['Scale','ACC', 'F1'], columns=['I
```

In [85]:
```
# 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)+"="*60)
    y = df[_type].values
    if method == 'RF':
        y = y.reshape(-1, 1)

    X_train, X_test, y_train, y_test = train_test_split(tfidf_df.values, y, test_size=
                                                        random_state=None, shuffle=Tru

    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 = LinearSVC(tol=1e-5)
        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)/l
        benchmark_df.loc['Scale', _type] = "{}% : {}%".format(int(sum(y)/len(y)*100+0.
        time.sleep(0.5)
        pred_train = [clf.predict(i.reshape(1,-1))[0] for i in tqdm(X_train)]
        acc = accuracy_score(y_train, pred_train)
        print("Accuracy on training set - %s" % _type, acc)
        benchmark_df.loc['ACC', _type] = acc
        f1 = f1_score(y_train, pred_train) if len(set(pred_train))>1 else 0.0
        print("F1 Score on training set - %s" % _type, f1)
        benchmark_df.loc['F1', _type] = f1
        time.sleep(0.5)
        pred_test = [clf.predict(i.reshape(1,-1))[0] 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 [86]:

```
# Separately benchmarking
clf_ie = train_by_type('IE', 'SVM', benchmark=True)
clf_ns = train_by_type('NS', 'SVM', benchmark=True)
clf_tf = train_by_type('TF', 'SVM', benchmark=True)
clf_jp = train_by_type('JP', 'SVM', benchmark=True)
```

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

# @Scoring START # --- SVM
Type: IE: I : E = 0.2304322766570605 : 0.7695677233429394
100%|██████████| 6940/6940 [00:00<00:00, 19087.43it/s]
Accuracy on training set - IE 0.8181556195965418
F1 Score on training set - IE 0.4793729372937293
100%|██████████| 1735/1735 [00:00<00:00, 15235.15it/s]
Accuracy on testing set - IE 0.7757925072046109
F1 Score on testing set - IE 0.367479674796748
# @Scoring END #

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

# @Scoring START # --- SVM
Type: NS: N : S = 0.13798270893371758 : 0.8620172910662824
100%|██████████| 6940/6940 [00:00<00:00, 19546.35it/s]
Accuracy on training set - NS 0.8821325648414986
F1 Score on training set - NS 0.3091216216216216
100%|██████████| 1735/1735 [00:00<00:00, 16004.96it/s]
Accuracy on testing set - NS 0.8553314121037464
F1 Score on testing set - NS 0.14915254237288134
# @Scoring END #

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

# @Scoring START # --- SVM
Type: TF: T : F = 0.5410951008645534 : 0.45890489913544663
100%|██████████| 6940/6940 [00:00<00:00, 19637.53it/s]
Accuracy on training set - TF 0.822478386167147
F1 Score on training set - TF 0.8369507676019058
100%|██████████| 1735/1735 [00:00<00:00, 15613.82it/s]
Accuracy on testing set - TF 0.7337175792507205
F1 Score on testing set - TF 0.756842105263158
# @Scoring END #

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

# @Scoring START # --- SVM
Type: JP: J : P = 0.604149855907781 : 0.39585014409221897
100%|██████████| 6940/6940 [00:00<00:00, 18876.07it/s]
Accuracy on training set - JP 0.7376080691642651
F1 Score on training set - JP 0.795554058605591
100%|██████████| 1735/1735 [00:00<00:00, 16537.02it/s]
Accuracy on testing set - JP 0.6334293948126801
F1 Score on testing set - JP 0.7158176943699731
# @Scoring END #
```

For the whole dataset, combine classifiers above and benchmark.

In [37]:
```
# predict_full_type
# @params:
#       text: (string), text(post) to predict.
#      _type: (string or None), True type; types:[INTJ~ESFP],
#                    if None: predict a non-recorde sample.
#     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=None, strict=True):
#     text = postVectorizer(clean_posts(text))
    IE = clf_ie.predict(text)[0]
    NS = clf_ns.predict(text)[0]
    TF = clf_tf.predict(text)[0]
    JP = clf_jp.predict(text)[0]
    match_rate = 0
    pred_type = type_dict[IE][0] + type_dict[NS][1] + type_dict[TF][2] + type_dict[JP][3]

    if _type is None:
        return pred_type

    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 [80]:
```
# Overall benchmarking strictly

test_size = df.shape[0]
# tests = [postVectorizer(clean_posts(df.sep_posts[i])) for i in range(test_size)]

preds = Parallel(n_jobs=4, prefer="threads")\
(delayed(predict_full_type)(r.reshape(1,-1), df.type[i], strict=True) for i, r in tqdm(enumerate(tfidf_df.

acc = accuracy_score([i[1] for i in preds], [i[0] for i in preds])
f1 = f1_score([i[1] for i in preds], [i[0] for i in preds], average='macro')
print("Overall Accuracy: {}".format(acc))
print("Overall F1 Score: {}".format(f1))

benchmark_df.loc['Scale', 'Full Type(Strict)'] = np.nan
benchmark_df.loc['ACC', 'Full Type(Strict)'] = acc
benchmark_df.loc['F1', 'Full Type(Strict)'] = f1
```

```
8675it [00:02, 2978.93it/s]

Overall Accuracy: 0.42662824207492794
Overall F1 Score: 0.23008778769482272
```

In [81]:
```
# Overall benchmarking non-strictly
test_size = df.shape[0]

acc = Parallel(n_jobs=4, prefer="threads")\
(delayed(predict_full_type)(r.reshape(1,-1), df.type[i], strict=False) for i, r in tqdm(enumerate(tfidf_df

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

benchmark_df.loc['Scale', 'Full Type(Loose)'] = np.nan
benchmark_df.loc['ACC', 'Full Type(Loose)'] = acc
benchmark_df.loc['F1', 'Full Type(Loose)'] = np.nan
```

```
[100%]MatchedPredicted Type: INFP  |  True Type: INTP  |  [ 75%]MatchedPredicted Type: INTP  |  True Ty
pe: ENFP  |  [ 50%]Matched


Predicted Type: INFJ  |  True Type: INTJ  |  [ 75%]Matched
Predicted Type: ENTJ  |  True Type: ENTP  |  [ 75%]Matched
Predicted Type: INTP  |  True Type: INTJ  |  [ 75%]Matched

Predicted Type: INFP  |  True Type: ENTP  |  [ 50%]MatchedPredicted Type: INFJ  |  True Type: INFJ  |
[100%]Matched
Predicted Type: INTP  |  True Type: ISFP  |  [ 50%]Matched
Predicted Type: INTP  |  True Type: ENFP  |  [ 50%]MatchedPredicted 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: 0.8040057636887608
```

In [82]:
```
benchmark_df
```

Out[82]:

	IE	NS	TF	JP	Full Type(Strict)	Full Type(Loose)
Scale	23% : 77%	14% : 86%	54% : 46%	60% : 40%	NaN	NaN
ACC	0.823055	0.881988	0.82147	0.738329	0.426628	0.804006
F1	0.49465	0.321458	0.836823	0.796321	0.230088	NaN

NOTE

1. Unlike the f1-scores in RF, svm doesn't cause deviation, however, it's easy to find, the better the type is balanced in training data, the higher the f1-score is.
2. Svm seems to be stable in this task, many different parameter combinations perform almost the same.

---

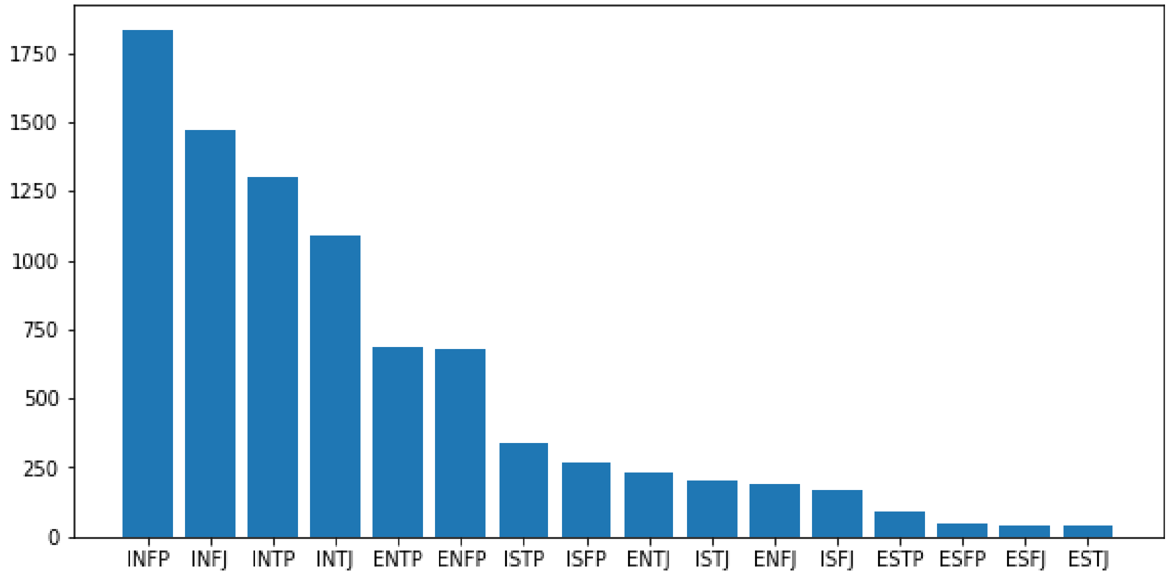
6 Further study

6.1 Introduction

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

In subtask1-RF, most of the posts were classified to INFP, which is the major type in training set.

`Distribution in Training set`



From Kaggle Forum dataset, I retrieved 36000 posts which have at least 1000 words.

Randomly, choose 200 posts from the dataset, and observe the distribution.

3 Further study on Kaggle ForumMessage

In [57]:
```
k_data = pd.read_csv('../dataset/ForumMessages.csv').Message.dropna().values
k_texts = []
for i in k_data:
    if len(i) > 1000:
        k_texts.append(i)
```

In [58]:
```
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 [94]:
```
cleaned_posts = [clean_posts(i) for i in k_texts]
res_counter = {}
```

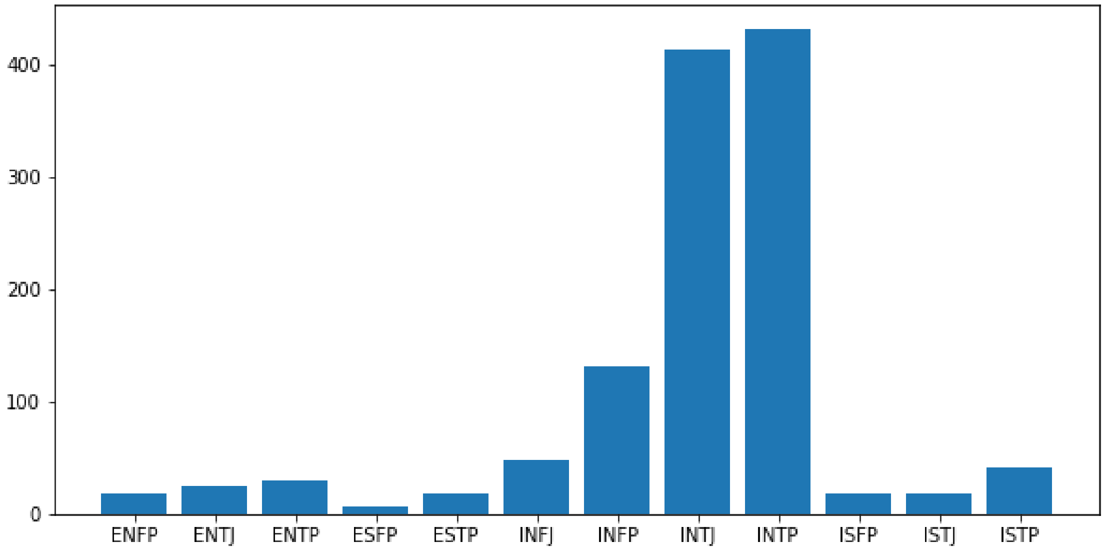
In [99]:
```
vectorized_posts = [postVectorizer(i) for i in random.sample(cleaned_posts, 200)]

res = Parallel(n_jobs=4, prefer="threads")\
(delayed(predict_full_type)(i, strict=True) for i in tqdm(vectorized_posts))
```

...

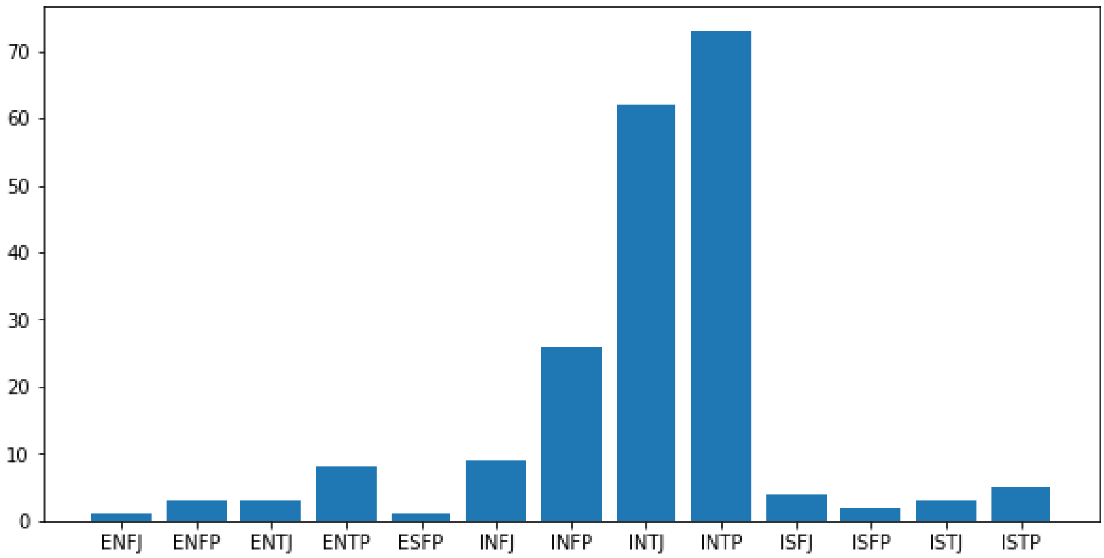
In [103]:
```
for i in res:
    if i not in res_counter.keys():
        res_counter[i] = 1
    else:
        res_counter[i] += 1

plt.figure(figsize=(10,5))
plt.bar(res_counter.keys(), [i for i in res_counter.values()])
```

6.3 Testcases

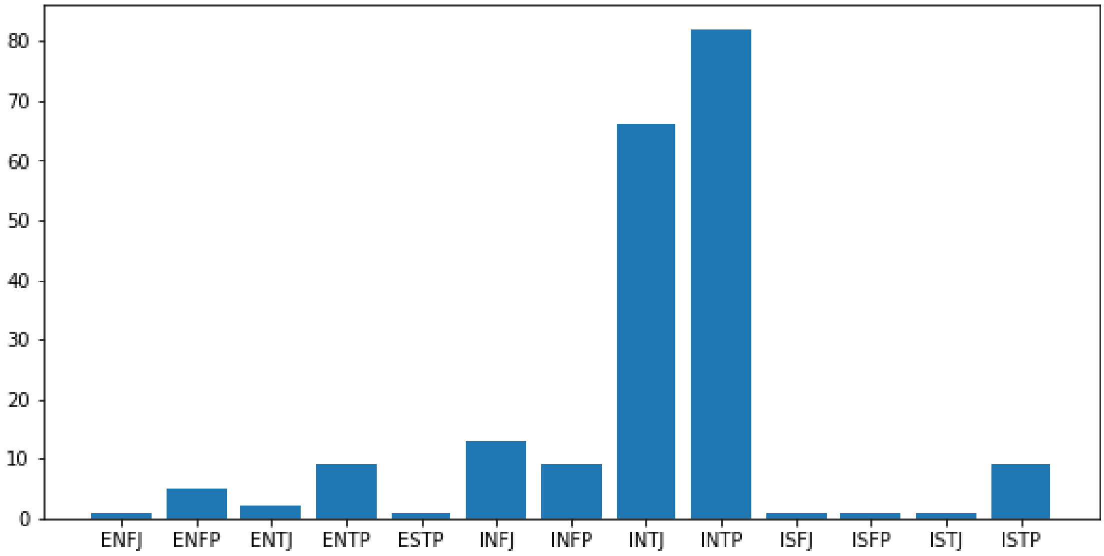
- testcase1



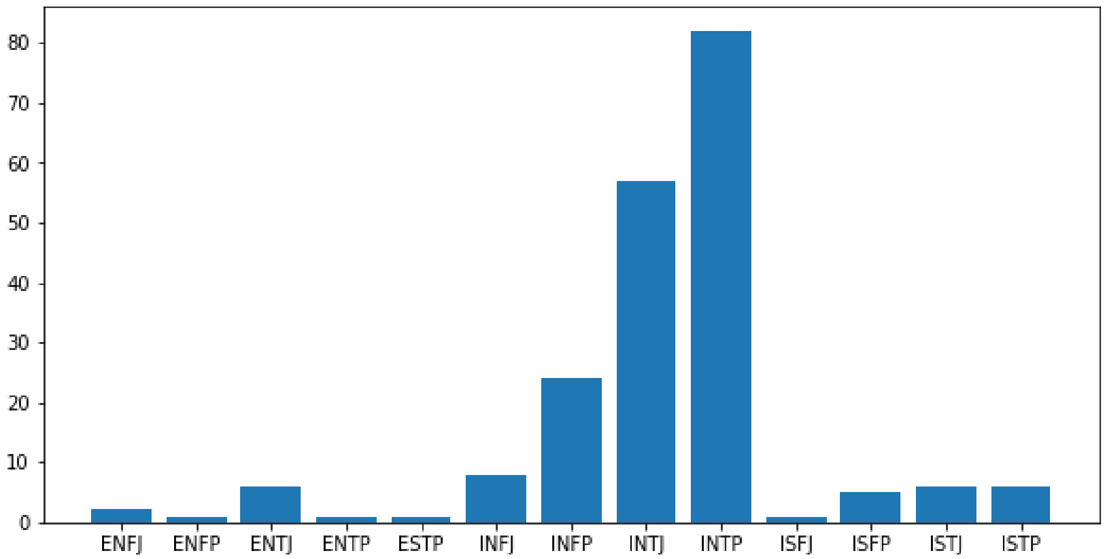
- testcase2



- testcase3



- testcase4



Although in each testcase, type INTJ and INTP took major places, other types appeared anyway.

7 Conclusion

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

1. src/mbti-SVM.ipynb
2. src/Preprocessor.ipynb
3. src/utilities.py

for a better experience!

Last but not least, take a look at the result below:

`Separately Benchmarking`

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

# @Scoring START # --- SVM
Type: IE: I : E = 0.2304322766570605 : 0.7695677233429394
100%|██████████| 6940/6940 [00:00<00:00, 19087.43it/s]
Accuracy on training set - IE 0.8181556195965418
F1 Score on training set - IE 0.4793729372937293
100%|██████████| 1735/1735 [00:00<00:00, 15235.15it/s]
Accuracy on testing set - IE 0.7757925072046109
F1 Score on testing set - IE 0.367479674796748
# @Scoring END #
```

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

# @Scoring START # --- SVM
Type: NS: N : S = 0.13798270893371758 : 0.8620172910662824
100%|██████████| 6940/6940 [00:00<00:00, 19546.35it/s]
Accuracy on training set - NS 0.8821325648414986
F1 Score on training set - NS 0.3091216216216216
100%|██████████| 1735/1735 [00:00<00:00, 16004.96it/s]
Accuracy on testing set - NS 0.8553314121037464
F1 Score on testing set - NS 0.14915254237288134
# @Scoring END #
```

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

# @Scoring START # --- SVM
Type: TF: T : F = 0.5410951008645534 : 0.45890489913544663
100%|██████████| 6940/6940 [00:00<00:00, 19637.53it/s]
Accuracy on training set - TF 0.822478386167147
F1 Score on training set - TF 0.8369507676019058
100%|██████████| 1735/1735 [00:00<00:00, 15613.82it/s]
Accuracy on testing set - TF 0.7337175792507205
F1 Score on testing set - TF 0.756842105263158
# @Scoring END #
```

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

# @Scoring START # --- SVM
Type: JP: J : P = 0.604149855907781 : 0.39585014409221897
100%|██████████| 6940/6940 [00:00<00:00, 18876.07it/s]
Accuracy on training set - JP 0.7376080691642651
F1 Score on training set - JP 0.795554058605591
100%|██████████| 1735/1735 [00:00<00:00, 16537.02it/s]
Accuracy on testing set - JP 0.6334293948126801
F1 Score on testing set - JP 0.7158176943699731
# @Scoring END #
```

`Overall Benchmarking`

	IE	NS	TF	JP	Full Type(Strict)	Full Type(Loose)
Scale	23% : 77%	14% : 86%	54% : 46%	60% : 40%	NaN	NaN
ACC	0.823055	0.881988	0.82147	0.738329	0.426628	0.804006
F1	0.49465	0.321458	0.836823	0.796321	0.230088	NaN

8 References

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

Karl 2020/6

BACK TO TOP