Ghouls, Goblins, and Ghosts... Boo!

This is a fun Halloween competition. We have some characteristics of monsters and the goal is to predict the type of monsters: ghouls, goblins or ghosts.

At first I do data exploration to get some insights. Then I try various models for prediction. The final prediction is done with the help of ensemble and majority voting.

In [1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set_style('whitegrid')

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV, Stratifi
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LogisticRegression
from sklearn import svm
import xgboost as xgb
from sklearn.ensemble import VotingClassifier
from sklearn.naive_bayes import GaussianNB
```

Data exploration

In [2]:
```
train = pd.read_csv('../input/train.csv')
test = pd.read_csv('../input/test.csv')
```

In [3]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 371 entries, 0 to 370
Data columns (total 7 columns):
id               371 non-null int64
bone_length      371 non-null float64
rotting_flesh    371 non-null float64
hair_length      371 non-null float64
has_soul         371 non-null float64
color            371 non-null object
type             371 non-null object
dtypes: float64(4), int64(1), object(2)
memory usage: 20.4+ KB
```

So there are 4 numerical variables and 1 categorical. And no missing values, which is nice!

In [4]:
```
train.describe(include='all')
```

Out[4]:

	id	bone_length	rotting_flesh	hair_length	has_soul	color
count	371.000000	371.000000	371.000000	371.000000	371.000000	371
unique	NaN	NaN	NaN	NaN	NaN	6
top	NaN	NaN	NaN	NaN	NaN	white
freq	NaN	NaN	NaN	NaN	NaN	137
mean	443.676550	0.434160	0.506848	0.529114	0.471392	NaN
std	263.222489	0.132833	0.146358	0.169902	0.176129	NaN
min	0.000000	0.061032	0.095687	0.134600	0.009402	NaN
25%	205.500000	0.340006	0.414812	0.407428	0.348002	NaN
50%	458.000000	0.434891	0.501552	0.538642	0.466372	NaN
75%	678.500000	0.517223	0.603977	0.647244	0.600610	NaN
max	897.000000	0.817001	0.932466	1.000000	0.935721	NaN

Numerical columns are either normalized or show a percentage, so no need to scale them.

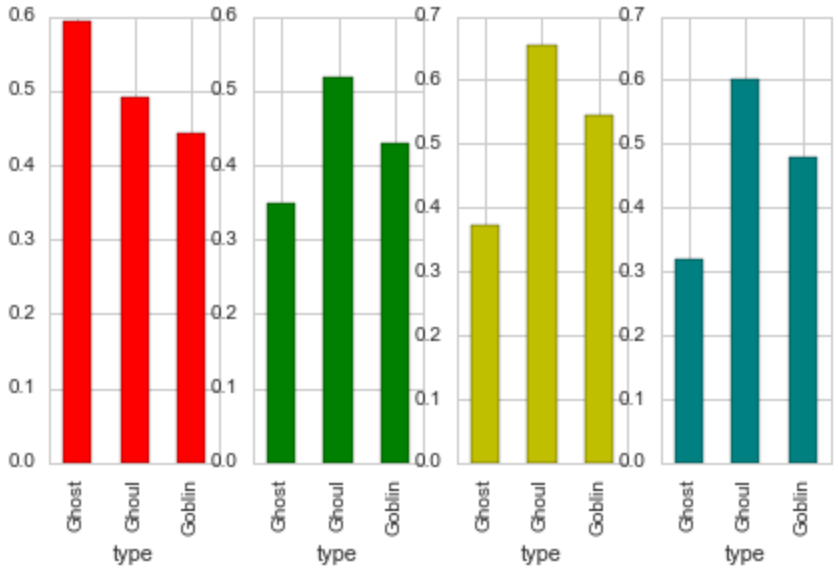
In [5]:
```
train.head()
```

Out[5]:

	id	bone_length	rotting_flesh	hair_length	has_soul	color	type
0	0	0.354512	0.350839	0.465761	0.781142	clear	Ghoul
1	1	0.575560	0.425868	0.531401	0.439899	green	Goblin
2	2	0.467875	0.354330	0.811616	0.791225	black	Ghoul
3	4	0.776652	0.508723	0.636766	0.884464	black	Ghoul
4	5	0.566117	0.875862	0.418594	0.636438	green	Ghost

In [6]:
```
plt.subplot(1,4,1)
train.groupby('type').mean()['rotting_flesh'].plot(kind='bar',figsize=(7,4),
plt.subplot(1,4,2)
train.groupby('type').mean()['bone_length'].plot(kind='bar',figsize=(7,4), c
plt.subplot(1,4,3)
train.groupby('type').mean()['hair_length'].plot(kind='bar',figsize=(7,4), c
plt.subplot(1,4,4)
train.groupby('type').mean()['has_soul'].plot(kind='bar',figsize=(7,4), colo
```

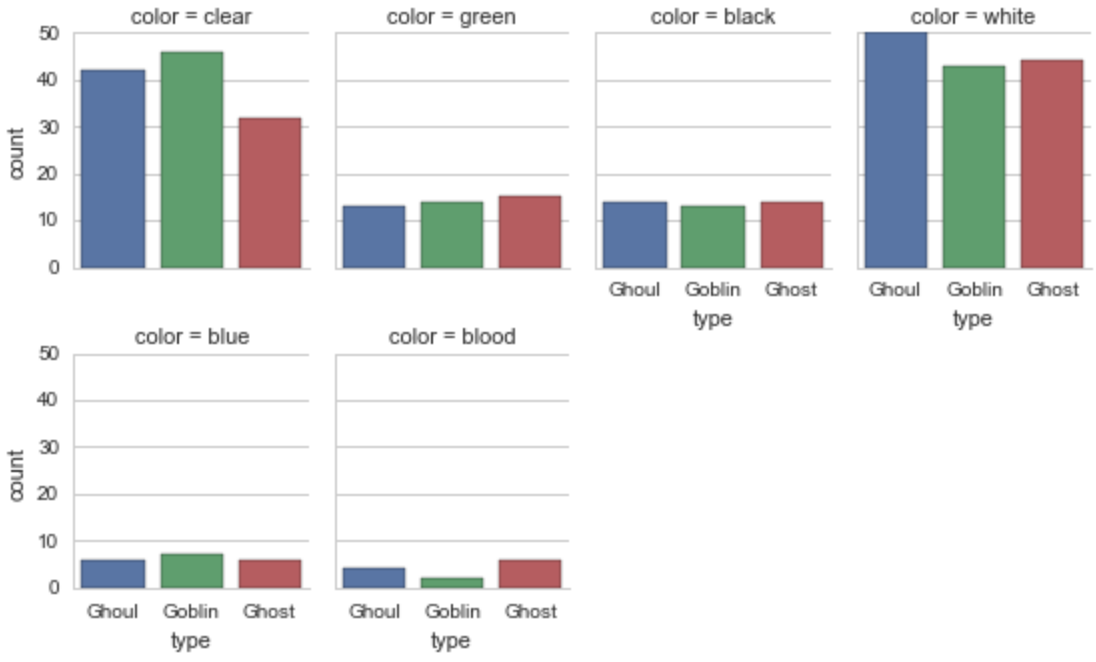
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x20152801d30>



It seems that all numerical features may be useful.

In [7]:
```
sns.factorplot("type", col="color", col_wrap=4, data=train, kind="count", si
```

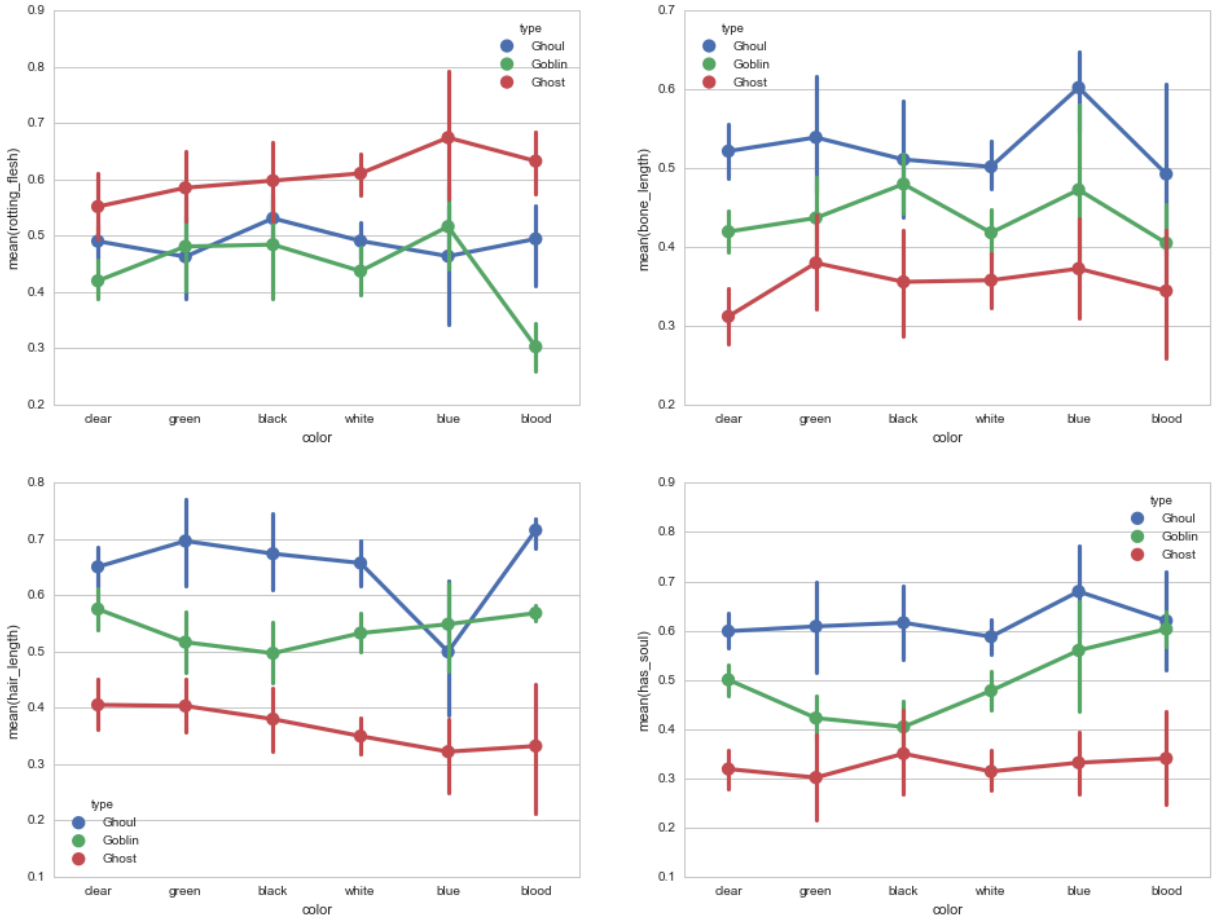
Out[7]: <seaborn.axisgrid.FacetGrid at 0x20152723898>



Funny, but many colors are evenly distributes among the monsters. So they maybe nor very useful for analysis.

In [8]:
```
fig, ax = plt.subplots(2, 2, figsize = (16, 12))
sns.pointplot(x="color", y="rotting_flesh", hue="type", data=train, ax = ax[
sns.pointplot(x="color", y="bone_length", hue="type", data=train, ax = ax[0,
sns.pointplot(x="color", y="hair_length", hue="type", data=train, ax = ax[1,
sns.pointplot(x="color", y="has_soul", hue="type", data=train, ax = ax[1, 1]
```

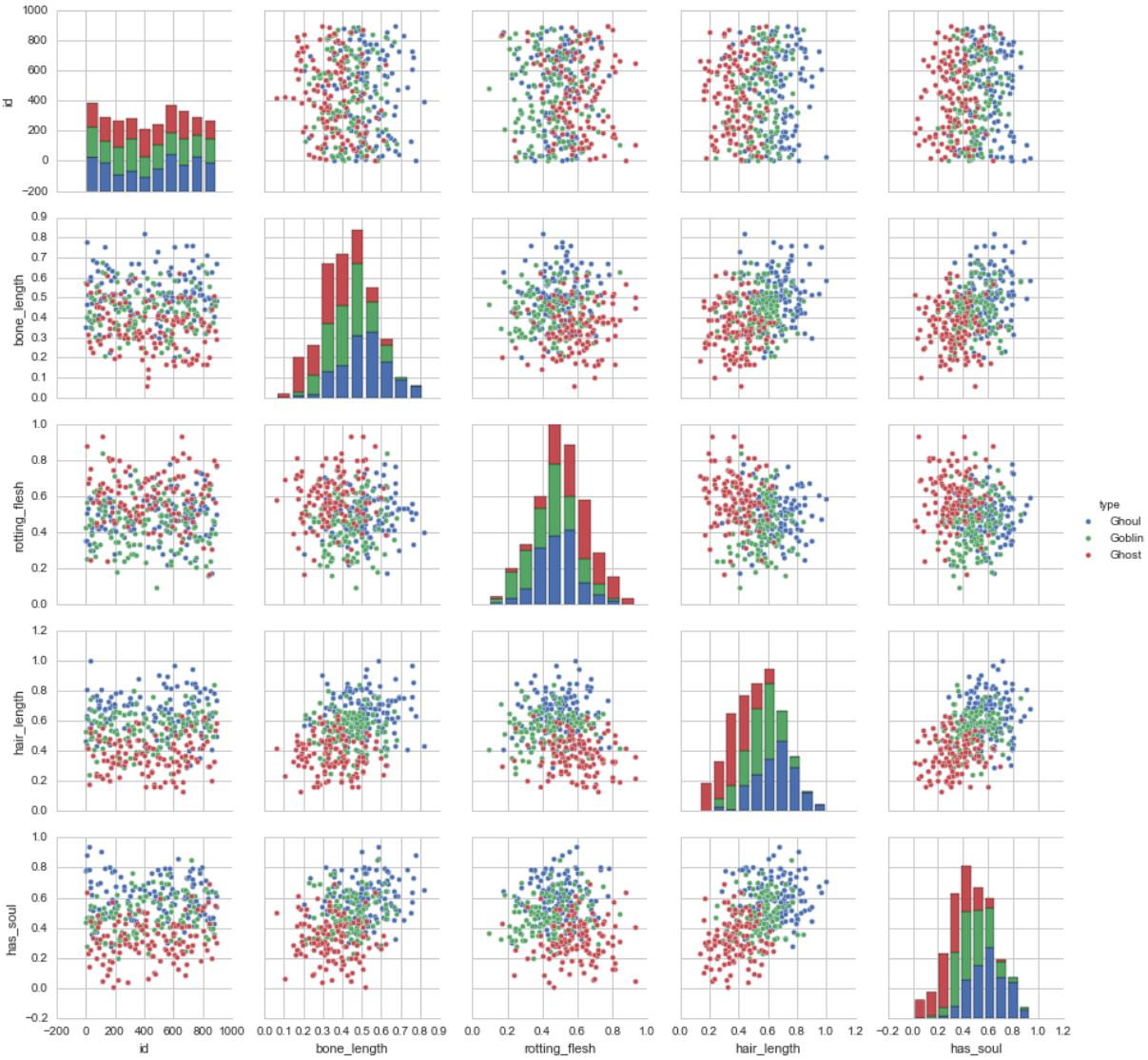
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x20152fa4710>



In most cases color won't "help" other variables to improve accuracy.

In [9]:
```
sns.pairplot(train, hue='type')
```

Out[9]: <seaborn.axisgrid.PairGrid at 0x20152e5a5f8>



This pairplot shows that data is distributed normally. And while most pairs are widely scattered (in relationship to the type), some of them show clusters: hair_length and has_soul, hair_length and bone_length. I decided to create new variables with multiplication of these columns and it worked great!

Data preparation

In [10]:
```
train['hair_soul'] = train['hair_length'] * train['has_soul']
train['hair_bone'] = train['hair_length'] * train['bone_length']
test['hair_soul'] = test['hair_length'] * test['has_soul']
test['hair_bone'] = test['hair_length'] * test['bone_length']
train['hair_soul_bone'] = train['hair_length'] * train['has_soul'] * train['
test['hair_soul_bone'] = test['hair_length'] * test['has_soul'] * test['bone
```

In [11]:
```
#test_id will be used later, so save it
test_id = test['id']
train.drop(['id'], axis=1, inplace=True)
test.drop(['id'], axis=1, inplace=True)
```

In [12]:
```
col = 'color'
dummies = pd.get_dummies(train[col], drop_first=False)
dummies = dummies.add_prefix("{}_".format(col))
train.drop(col, axis=1, inplace=True)
train = train.join(dummies)
dummies = pd.get_dummies(test[col], drop_first=False)
dummies = dummies.add_prefix("{}_".format(col))
test.drop(col, axis=1, inplace=True)
test = test.join(dummies)
```

In [13]:
```
X_train = train.drop('type', axis=1)
le = LabelEncoder()
Y_train = le.fit_transform(train.type.values)
X_test = test
```

In [14]:
```
clf = RandomForestClassifier(n_estimators=200)
clf = clf.fit(X_train, Y_train)
indices = np.argsort(clf.feature_importances_)[::-1]

print('Feature ranking:')
for f in range(X_train.shape[1]):
    print('%d. feature %d %s (%f)' % (f + 1, indices[f], X_train.columns[inc
                                      clf.feature_importances_[indices[f]]))
```

```
Feature ranking:
1. feature 6 hair_soul_bone (0.200967)
2. feature 4 hair_soul (0.166553)
3. feature 5 hair_bone (0.136124)
4. feature 2 hair_length (0.130899)
5. feature 1 rotting_flesh (0.119869)
6. feature 3 has_soul (0.116268)
7. feature 0 bone_length (0.088514)
8. feature 10 color_clear (0.009504)
9. feature 12 color_white (0.009176)
10. feature 9 color_blue (0.007161)
11. feature 11 color_green (0.006741)
12. feature 7 color_black (0.006363)
13. feature 8 color_blood (0.001860)
```

Graphs and model show that color has little impact, so I won't use it. In fact I tried using it, but the result got worse. And three features, which I created, seem to be important!

In [15]:
```
best_features = X_train.columns[indices[0:7]]
X = X_train[best_features]
Xt = X_test[best_features]
```

Model

In [16]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, Y_train, test_size=0.20,
```

Tune the model. Normally you input all parameters and their potential values and run GridSearchCV. My PC isn't good enough so I divide parameters in two groups and repeatedly run two GridSearchCV until I'm satisfied with the result. This gives a balance between the quality and the speed.

In [17]:
```
forest = RandomForestClassifier(max_depth = 100,
                                min_samples_split =2,
                                min_weight_fraction_leaf = 0.0,
                                max_leaf_nodes = 40)

parameter_grid = {'n_estimators' : [10, 20, 150],
                  'criterion' : ['gini', 'entropy'],
                  'max_features' : ['auto', 'sqrt', 'log2']
                 }

grid_search = GridSearchCV(forest, param_grid=parameter_grid, scoring='accur
grid_search.fit(X, Y_train)
print('Best score: {}'.format(grid_search.best_score_))
print('Best parameters: {}'.format(grid_search.best_params_))
```

```
Best score: 0.7035040431266847
Best parameters: {'max_features': 'log2', 'n_estimators': 20, 'criterion':
'gini'}
```

In [18]:
```
forest = RandomForestClassifier(n_estimators = 20,
                                criterion = 'entropy',
                                max_features = 'sqrt')
parameter_grid = {
                  'max_depth' : [None, 5, 100],
                  'min_samples_split' : [2, 5, 7],
                  'min_weight_fraction_leaf' : [0.0, 0.1],
                  'max_leaf_nodes' : [40, 80],
                 }

grid_search = GridSearchCV(forest, param_grid=parameter_grid, scoring='accur
grid_search.fit(X, Y_train)
print('Best score: {}'.format(grid_search.best_score_))
print('Best parameters: {}'.format(grid_search.best_params_))
```

```
Best score: 0.7277628032345014
Best parameters: {'max_leaf_nodes': 40, 'max_depth': 5, 'min_weight_fraction
_leaf': 0.1, 'min_samples_split': 5}
```

Calibrated classifier gives probabilities for each class, so to check the accuracy at first I chose the most probable class and convert it to values. Then I compare it to values of validation set.

In [19]:
```
#Optimal parameters
clf = RandomForestClassifier(n_estimators=20, n_jobs=-1, criterion = 'gini',
                             min_samples_split=2, min_weight_fraction_leaf=0
                             max_leaf_nodes=40, max_depth=100)

calibrated_clf = CalibratedClassifierCV(clf, method='sigmoid', cv=5)
calibrated_clf.fit(Xtrain, ytrain)
```

```
y_val = calibrated_clf.predict_proba(Xtest)

print("Validation accuracy: ", sum(pd.DataFrame(y_val, columns=le.classes_).
                                   == le.inverse_transform(ytest))/len(ytest
```

```
Validation accuracy:  0.653333333333
```

I used the best parameters and validation accuracy is ~68-72%. Not bad. But let's try something else.

In [20]:
```
svc = svm.SVC(kernel='linear')
svc.fit(Xtrain, ytrain)
y_val_s = svc.predict(Xtest)
print("Validation accuracy: ", sum(le.inverse_transform(y_val_s) == le.inver
```

```
Validation accuracy:  0.76
```

Much better! Usually RandomForest requires a lot of data for good performance. It seems that in this case there was too little data for it.

In [21]:
```
#The last model is logistic regression
logreg = LogisticRegression()

parameter_grid = {'solver' : ['newton-cg', 'lbfgs'],
                  'multi_class' : ['ovr', 'multinomial'],
                  'C' : [0.005, 0.01, 1, 10, 100, 1000],
                  'tol': [0.0001, 0.001, 0.005]
                 }

grid_search = GridSearchCV(logreg, param_grid=parameter_grid, cv=StratifiedK
grid_search.fit(Xtrain, ytrain)
print('Best score: {}'.format(grid_search.best_score_))
print('Best parameters: {}'.format(grid_search.best_params_))
```

```
Best score: 0.75
Best parameters: {'multi_class': 'multinomial', 'C': 1, 'tol': 0.0001, 'solv
er': 'newton-cg'}
```

In [22]:
```
log_reg = LogisticRegression(C = 1, tol = 0.0001, solver='newton-cg', multi_
log_reg.fit(Xtrain, ytrain)
y_val_l = log_reg.predict_proba(Xtest)
print("Validation accuracy: ", sum(pd.DataFrame(y_val_l, columns=le.classes_
                                   == le.inverse_transform(ytest))/len(ytest
```

```
Validation accuracy:  0.773333333333
```

It seems that regression is better. The reason? As far as I understand, the algorithms are similar, but with different loss function. And most importantly: SVC is a hard classifier while LR gives probabilities.

And then I received an advice to try ensemble or voting. Let's see.

Voting can be done manually or with sklearn classifier. For manual voting I need to make predictions for each classifier and to take the most common one. Advantage is that I may use any classifier I want, disadvantage is that I need to

do it manually. Also, if some classifiers give predictions as classed and others as probability distribution, it complicates things. Or I can use sklearn.ensemble.VotingClassifier. Advantage is that it is easier to use. Disadvantage is that it may use only sklearn algorithms (or more precisely - algorithms with method "get_param") and only those which can give probability predictions (so no SVC and XGBooost). Well, SVC can be used if correct parameters are set. I tried both ways and got the same accuracy as a result.

In [23]:
```
clf = RandomForestClassifier(n_estimators=20, n_jobs=-1, criterion = 'gini',
                             min_samples_split=2, min_weight_fraction_leaf=0
                             max_leaf_nodes=40, max_depth=100)

calibrated_clf = CalibratedClassifierCV(clf, method='sigmoid', cv=5)
log_reg = LogisticRegression(C = 1, tol = 0.0001, solver='newton-cg', multi_
gnb = GaussianNB()
```

In [24]:
```
calibrated_clf1 = CalibratedClassifierCV(RandomForestClassifier())
log_reg1 = LogisticRegression()
gnb1 = GaussianNB()
```

As far as I can understand, while using hard voting, it is better to used unfitted estimators. Hard voting uses predicted class labels for majority rule voting. Soft voting predicts the class label based on the argmax of the sums of the predicted probalities, which is recommended for an ensemble of well-calibrated classifiers. And with soft voting we can use weights for models.

In [25]:
```
Vclf1 = VotingClassifier(estimators=[('LR', log_reg1), ('CRF', calibrated_cl
                                     ('GNB', gnb1)], voting='hard')
Vclf = VotingClassifier(estimators=[('LR', log_reg), ('CRF', calibrated_clf)
                                     ('GNB', gnb)], voting='soft', weights=[
```

In [26]:
```
hard_predict = le.inverse_transform(Vclf1.fit(X, Y_train).predict(Xt))
soft_predict = le.inverse_transform(Vclf.fit(X, Y_train).predict(Xt))
```

In [27]:
```
#Let's see the differences:
for i in range(len(hard_predict)):
    if hard_predict[i] != soft_predict[i]:
        print(i, hard_predict[i], soft_predict[i])
```

```
3 Ghost Goblin
24 Ghoul Goblin
40 Goblin Ghost
44 Goblin Ghost
51 Ghoul Goblin
71 Ghost Goblin
74 Ghoul Goblin
93 Goblin Ghost
100 Ghoul Goblin
111 Goblin Ghost
120 Ghoul Goblin
123 Ghoul Goblin
137 Ghoul Goblin
152 Goblin Ghoul
211 Ghost Goblin
230 Goblin Ghost
238 Ghoul Goblin
254 Ghoul Goblin
273 Goblin Ghost
299 Ghoul Goblin
300 Ghoul Goblin
305 Ghoul Goblin
316 Goblin Ghost
338 Ghoul Goblin
378 Goblin Ghost
393 Goblin Ghost
398 Ghost Goblin
411 Ghost Goblin
418 Ghoul Goblin
431 Ghost Goblin
445 Ghoul Goblin
453 Ghoul Goblin
```

There are differences, but both predictions give the same result on leaderboard. I think that some ensemble of voting classifiers could improve score. For example use different classifiers for several VotingClassifiers and them make a majority voting on these VotingClassifiers.

In [28]:
```
submission = pd.DataFrame({'id':test_id, 'type':hard_predict})
submission.to_csv('GGG_submission3.csv', index=False)
```

The competition has started some time ago. My LR model got 0.73346 and majority voting 0.74291, which was at top 10% at that moment. Currently top accuracy is 0.74858. After some tweaking my voting got 0.74480.