SD 210 : Report of the data challenge

Student: YAHMED Taycir

Description

This document holds the approach I used in the data challenge. To reproduce the results, you can check the README file (how to run the different python scripts). I opted for a separate report rather than an ipython format because the scripts take so long to run, which in my understanding makes the notebook useless when checking the different solutions. Please note however that this report contains sections where I illustrate the code I used to implement some ideas.

The used packages

In [2]:

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LogisticRegression , LinearRegression
from sklearn.metrics import roc_auc_score , make_scorer
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
import pickle
from sklearn.ensemble import ExtraTreesClassifier , BaggingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier ,  VotingClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.gaussian_process.kernels import RBF
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.svm import OneClassSVM
from sklearn.covariance import EllipticEnvelope
from sklearn import decomposition
from scipy import stats
from sklearn.preprocessing import normalize
import matplotlib.pyplot as plt
from scipy import signal
from sklearn.cluster import KMeans
from sklearn import preprocessing
import operator
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn.preprocessing import normalize
import matplotlib.pyplot as plt
from scipy import signal
from sklearn.cluster import KMeans
from sklearn import preprocessing
from IPython.display import Image
```

Loading data

The first thing to do is for sure loading the data. For that I used the following code.

In [3]:

```
X_train_fname = './data/training_templates.csv'
y_train_fname = './data/training_labels.txt'
X_test_fname  = './data/testing_templates.csv'
X_train = pd.read_csv(X_train_fname, sep=',', header=None).values
X_test  = pd.read_csv(X_test_fname,  sep=',', header=None).values
y_train = np.loadtxt(y_train_fname, dtype=np.int)
```

Note :

Before starting to do any data analytics, we want to be sure about using the right scoring metric. In this challenge, we have a non-traditional metric. So we have to define a scoring function that can be adapted to model selection and evaluation tools (such as GridSearchCV and cross_val_score, ..) and that controls what value to apply in the estimators evaluation.

In [4]:

```
def compute_pred_score(y_true, y_pred):
    y_pred_unq =  np.unique(y_pred)
    for i in y_pred_unq:
        if((i != -1) & (i!= 1) & (i!= 0) ):
            raise ValueError('The predictions can contain only -1, 1, or 0!')
    y_comp = y_true * y_pred
    score = float(10*np.sum(y_comp == -1) + np.sum(y_comp == 0))
    score /= y_comp.shape[0]
    return score

loss  = make_scorer(compute_pred_score, greater_is_better=False)
```

Note that we put the attribute "greater_is_better" to false, because we want to minimize this value. Indeed, this attribute indicates whether the function is a score function (default), meaning high is good, or a loss function, meaning low is good. In the latter case, the scorer object will sign-flip the outcome of the function 'compute_pred_score'.

Preprocessing data

The set of techniques used prior to the application of a data mining method is named as data preprocessing for data mining and it is known to be one of the most meaningful issues within the famous Knowledge Discovery from Data process. Since data will likely be imperfect, containing inconsistencies and redundancies is not directly applicable for a starting a data mining process. Preprocessing data is certainly a major step in any data challenge.

To preprocess the provided data, I tried doing tasks like:

- **denoising** : I did some research on image denoising, also related to the SD205 course where at some point we work on image processing and reducing noise. It seemed to me obvious to do so, since we are dealing with an image challenge. I first tried denoising with the Wiener filter.

In [5]:

```
X_train = signal.wiener(X_train)
X_test = signal.wiener(X_test)
```

In signal processing, the Wiener filter is a filter used to produce an estimate of a desired or target random process by linear time-invariant (LTI) filtering of an observed noisy process, assuming known stationary signal and noise spectra, and additive noise. The Wiener filter minimizes the mean square error between the estimated random process and the desired process. The goal of the Wiener filter is to compute a statistical estimate of an unknown signal using a related signal as an input and filtering that known signal to produce the estimate as an output.

- **randomizing data** : When checking the training labels file, I noticed that the values are ordered : The first lines are all assigned to 1 and the last lines are all assigned to -1. So I thought one good suggestion would be randomizing the data to encourage the generalization of my models. I suspect some of them would be altered by the organized structure of the training data.

In [6]:

```
X = np.hstack((X_train , y_train.reshape(-1,1)))
np.random.shuffle(X)
m , n = X.shape
y_train = X[: , n-1]
X_train = np.delete(X, n-1 , 1)
```

- **scaling data** : Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn; they might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.In practice we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation. For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the l1 and l2 regularizers of linear models) assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected.

In [7]:

```
X_train = preprocessing.scale(X_train)
X_test = preprocessing.scale(X_test)
```

- **removing outliers** : Outliers are one of those statistical issues that everyone knows about, but most people aren’t sure how to deal with. Most parametric statistics, like means, standard deviations, and correlations, and every statistic based on these, are highly sensitive to outliers. And since the assumptions of common statistical procedures, like linear regression and ANOVA, are also based on these statistics, outliers can really mess up your analysis. Despite all this, as much as you’d like to, it is NOT acceptable to drop an observation just because it is an outlier. They can be legitimate observations and are sometimes the most interesting ones. It’s important to investigate the nature of the outlier before deciding.

In the following code I tried to remove outliers (I put a filter such that we select all rows where the values of a certain column are within say 3 standard deviations from mean), but I noticed that this reduces the performance, so I gave that up eventually.

In [11]:

```
X = np.hstack((X_train , y_train.reshape(-1,1)))
X = X[(np.abs(stats.zscore(X)) < 3).all(axis=1)]
m , n = X.shape
y_train = X[: , n-1]
X_train = np.delete(X, n-1 , 1)
```

Feature engineering

Reading different articles on how to win data challenges, I noticed that all of them state that feature engineering is the most important part of any data challenge. So, I considered doing some data wrangling, though I was really confused by the fact that we are working on features extracted from images (Sift maybe ?) not the images themselves. As a matter of fact, a lot of the articles I checked on image challenges suggested some nice/interesting methods to preprocess images in order to prepare them for the learning phase, but that I couldn't experiment due to the type of data we are given (numerical matrices). I tried however to implement some ideas, such as:

- **Principal component analysis (PCA)** Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

In [8]:

```
clf = decomposition.PCA(n_components= 3 , svd_solver='randomized',  whiten=True)
X_train = np.hstack((X_train , clf.fit_transform(X_train)))
X_test = np.hstack((X_test , clf.transform(X_test)))
```

- **Recursive feature elimination**: Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

I used RFECV : it performs RFE in a cross-validation loop to find the optimal number of features.

In [15]:

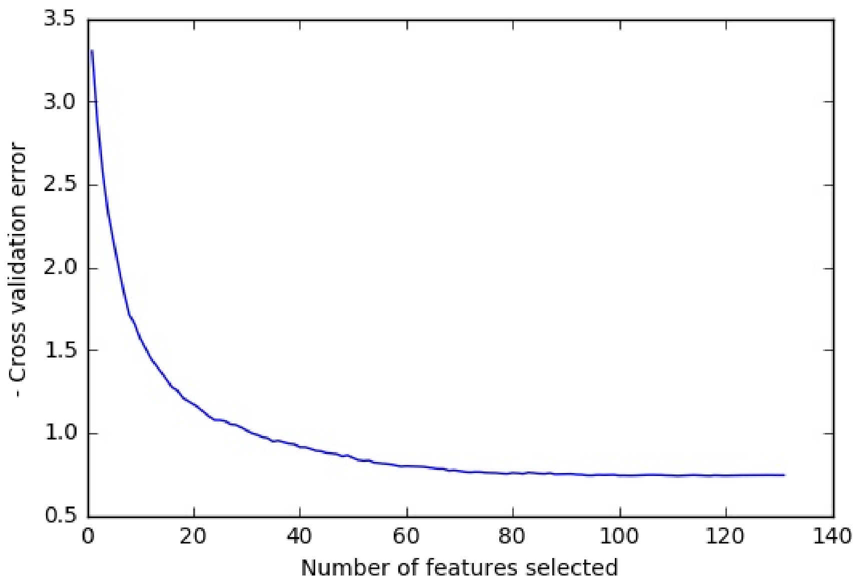
```
clf = LogisticRegression(C=100)
print -cross_val_score(clf, X_train, y_train, cv=5 , scoring = loss).mean()
rfecv = RFECV(estimator=clf, step=1, cv=StratifiedKFold(5),  scoring=loss)
X_train = rfecv.fit_transform(X_train, y_train)
X_test = rfecv.transform(X_test)

print("Optimal number of features : %d" % rfecv.n_features_)
```

```
0.742565748378
Optimal number of features : 111
```

In [17]:

```
# Plot number of features VS. cross-validation scores
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("- Cross validation error")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), -1*rfecv.grid_scores_)
plt.show()
```



Giving the previous results : I decided not to consider this approach because it was clearly not suggesting a promising dimensionnality reduction on the dataset.

- **clustering** : Another idea that I find quite recurrent in papers related to image/face classification/recognition is clustering the dataset and adding the results as another feature. It was also suggested by some professors at Telecom. To perform that, I used k-means: a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition and observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

In [10]:

```
kmeans = KMeans(n_clusters=2, random_state=0).fit(X_train)
kres_train = kmeans.labels_
kres_test = kmeans.predict(X_test)
X_train = np.hstack((X_train , kres_train.reshape(-1,1)))
X_test = np.hstack((X_test , kres_test.reshape(-1,1)))
```

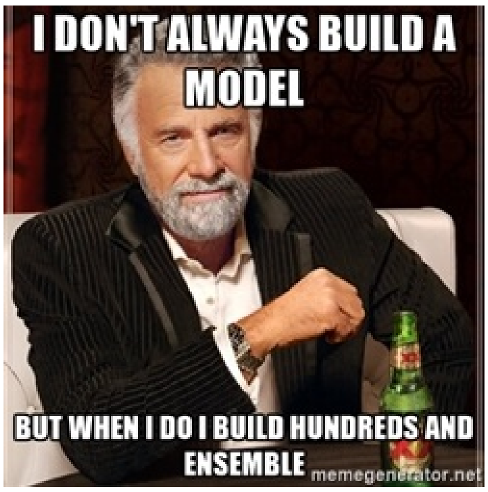
Learning phase

My approach is driven by ensembling different models.

In [57]:

```
Image( url="https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2017/02/12152925/meme.jpg"
, width = 300)
```

Out[57]:


I DON'T ALWAYS BUILD A MODEL
BUT WHEN I DO I BUILD HUNDREDS AND ENSEMBLE
memegenerator.net

I tried many ideas, such as:

- stacking, blending and averaging: (the first approach I used) I added to an initial set of 128 features, new features being the predictions of N different classifiers (Random Forest, GBM, Neural Networks,…). And then retrained P classifiers over the 128 + N features. And finally made a weighted average of the P outputs.
- I also tried running another classifier at the end instead of calculating the weighted average. Using these methods, the error didn't drop (at least compared to the performance other people got on the public leader board), even after tuning hyperparameters, so I tried a whole different approach.

In the following part of the report, I will present the ensembling method that gave me the best (least) error.

Step 1 : determine the best performing algorithms among the classifiers available in sklearn

Note:

I also tried the packages nolearn.lasagne and the xgboost implementation of Gradient Boosting but they didn't give better results.

In [59]:

```
names = [ "LogisticRegression", "RandomForestClassifier" , "GradientBoostingClassifier",
         "ExtraTreesClassifier", "SVC  linear", "SVC", "Neural Net", "AdaBoost",  " GaussianNB" , "QDA"]

classifiers = [
    LogisticRegression (),
    RandomForestClassifier(),
    GradientBoostingClassifier(),
    ExtraTreesClassifier(),
    SVC(probability = True),
    SVC(probability = True),
    MLPClassifier(),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis()]

loss  = make_scorer(compute_pred_score, greater_is_better=False)
w = []
for name, clf in zip(names, classifiers):
  print "start ", name
  wc = -1/learn (clf, X_train , y_train , loss)
  w.append(wc)
  print "end" , name

print zip(names , w)
```

The output :

LogisticRegression = 1.3714285714285712

RandomForestClassifier = 0.64980616577441397

GradientBoostingClassifier = 0.94268880557043389

ExtraTreesClassifier = 0.59439378588314762

SVC linear = 1.2764414359966154

SVC = 1.2764414359966154

Neural Net = 3.740701381509032

AdaBoost = 0.52707761417519339

GaussianNB = 0.72160721607216072

QDA = 1.6301327570237725

That's how I chose QuadraticDiscriminantAnalysis , MLPClassifier & LogisticRegression

(I also tried to add SVC -the 4th best performing algorithm- afterwards but it didn't improve the results)

Step 2 : Tuning the hyperparameters for the selected models

LogisticRegression :

In [19]:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000 , 10000 , 100000]}

clf = GridSearchCV(LogisticRegression (), param_grid, cv=5 , scoring = loss)
clf.fit(X_train, y_train)

print "params", clf.best_params_
print "best score", clf.best_score_

"""
params {'C': 100}
best score -0.719507575758
"""
```

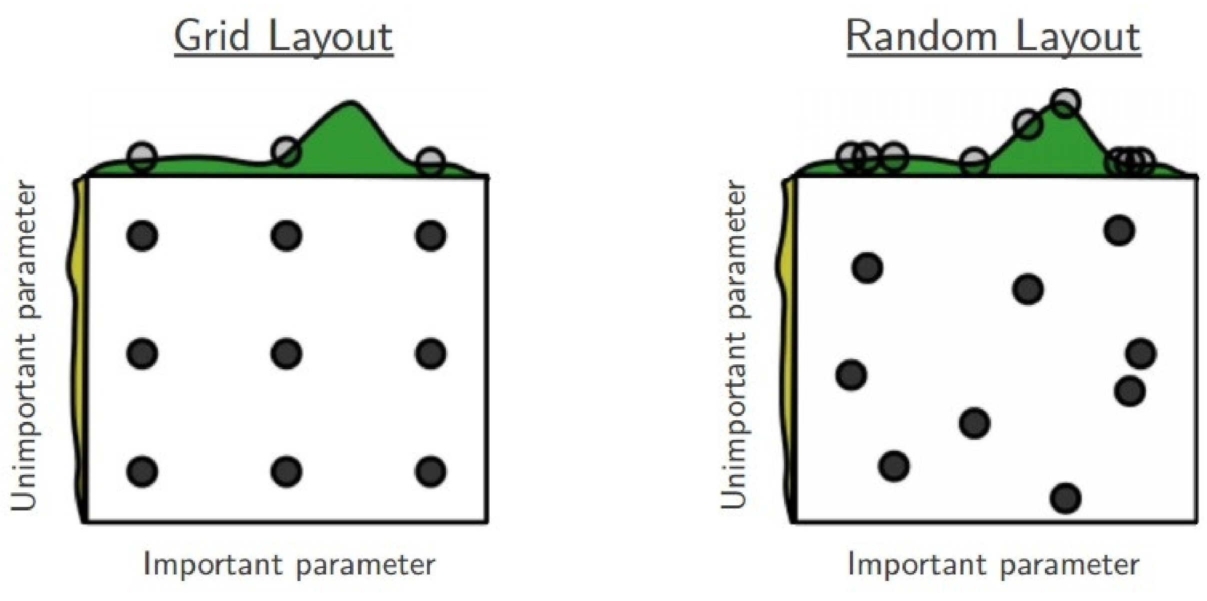
MLPClassifier:

The problem was to find a way to quickly find the best hyperparameters combination. I first discovered GridSearchCV, that makes an exhaustive search over specified parameter ranges. As always with scikit-learn, it has a convenient programming interface, handling for example smoothly cross-validation and parallel distributing of search. However, the number of parameters to tune, and their range, was too large to discover the best ones in the acceptable time frame I had in mind (typically while sleeping). I had to fall back to another option : I then used RandomizedSearchCV, that appeared in 0.14 version. With this method, search is done randomly on a subspace of parameters. It gives generally very good results and I was able to find a suitable parameter set within a few hours.

```
In [58]:
```

```
Image(url="http://5047-presscdn.pagely.netdna-cdn.com/wp-content/uploads/2015/07/scikitlearn8.jpeg")
```

```
Out[58]:
```



But I also discovered a nice way to perform a finer hyperparameters optimization in a reasonable time: After doing a random search, we can then iterate by “zooming in” on the regions of the hyperparameter space which look promising. We can do this by running additional, more targeted, random or Cartesian hyperparameter searches or manual searches. For example, if we started with alpha values of [0.00001 , 0.0001 , 0.001 , 0.01 , 0.1 , 1 , 10 , 100 , 1000] and get 0.01 with randomized grid search, we can follow up with a finer grid of [0.0001 , 0.001 , 0.01 , 0.1] with grid search.

```
In [20]:
```

```
# Randomized grid search
param_grid =  {'hidden_layer_sizes': [(nb,) for nb in range(20,500,50)] , 'alpha': [0.00001 , 0.0001 , 0.001
 , 0.01 , 0.1 , 1 , 10 , 100 , 1000]}
clf = RandomizedSearchCV(MLPClassifier(), param_grid, cv=5 , scoring = loss , n_iter = 20 )
clf.fit(X_train, y_train)

print "params", clf.best_params_
print "best score", -clf.best_score_
```

Output : params {'alpha': 0.01, 'hidden_layer_sizes': (370,)} best score 0.233143939394

```
In [23]:
```

```
# fine exaustive grid search
param_grid =  {'hidden_layer_sizes': [(nb,) for nb in range(280,400,20)] , 'alpha': [ 0.0001 , 0.001 , 0.01
, 0.1]}

clf = GridSearchCV(MLPClassifier(), param_grid, cv=5 , scoring = loss )
clf.fit(X_train, y_train)

print "params", clf.best_params_
print "best score", -clf.best_score_
```

Output : params {'alpha': 0.01, 'hidden_layer_sizes': (340,)} best score 0.232765151515

Step 3 : Use bagging and tune its hyperparameters

One of the secret of data competitions is to run several times the same algorithm, with random selection of observations and features, and take the average of the output. To do that easily, I discovered the scikit-learn BaggingClassifier meta-estimator. It hides the tedious complexity of looping over model fits, random subsets selection, and averaging—and exposes easy fit() / predict_proba() entry points.

For tuning the bagging hyperparameters, I only performed that on LogisticRegression and QuadraticDiscriminantAnalysis, because for the the MLPClassifier it's taking ages (more than my poor pc can handle..).

In [24]:

```
param_grid = {'max_features': [0.6, 0.8, 1.0], 'max_samples': [0.6, 0.8, 1.0]}
names = [  "LR",  "QDA", "Neural Net"]
classifiers = [
    GridSearchCV(BaggingClassifier(LogisticRegression (C = 100)), param_grid=param_grid , cv = 5 , scoring =
 loss),
    GridSearchCV(BaggingClassifier(QuadraticDiscriminantAnalysis()), param_grid=param_grid, cv = 5 , scoring
 = loss),
    ]

# LogisticRegression
clf = classifiers[0]
clf.fit(X_train , y_train)

print  clf.best_params_
# {'max_features': 1.0, 'max_samples': 0.6}

# QuadraticDiscriminantAnalysis
clf = classifiers[1]
clf.fit(X_train , y_train)

print clf.best_params_
#{'max_features': 0.6, 'max_samples': 0.8}
```

Note: I also tried to use calibration (instead of bagging) : This is one of the great functionalities of the last scikit-learn version (0.16). It allows to rescale the classifier predictions by taking observations predicted within a segments (e.g. 0.3–04), and comparing to the actual truth ratio of these observation (e.g. 0.23, with means that a rescaling is needed). But I noticed that it overfitted the dataset.

Step 4 : Voting classifier with adapted weights

Voting classifier is a Soft Voting/Majority Rule classifier for unfitted estimators. (I used the soft voting: predicts the class label based on the argmax of the sums of the predicted probabilities, which is recommended for an ensemble of well-calibrated classifiers).

First of all we determine the weights:

We consider the inverse of the error as the weight for each of the **tuned** classifiers. These errors are calculated with cross validation (cf 'learn' function)

In [27]:

```
def learn(clf , X_train , y_train , loss ):
  w = cross_val_score(clf, X_train, y_train, cv=5 , scoring = loss).mean()
  return w

names = [  "LR",  "QDA", "Neural Net"]
classifiers = [
    BaggingClassifier(LogisticRegression (C = 100) , max_samples=0.6, max_features=1),
    BaggingClassifier(QuadraticDiscriminantAnalysis(),  max_samples=0.8, max_features=0.6),
    BaggingClassifier(MLPClassifier(alpha=0.01, hidden_layer_sizes = 340) ,  max_samples=0.5, max_features=0
.5)
    ]
w = []
for name, clf in zip(names, classifiers):
  wc = -1/learn (clf, X_train , y_train , loss)
  w.append(wc)
```

Second : we run the voting classifier with the adapted weigths

In [29]:

```
clf = VotingClassifier(estimators=zip(names , classifiers), voting='soft' , weights=w)
clf.fit (X_train , y_train)
```

Third: Minimizing error

To do that, we look for a threshold (to compare with probabilities of the classes) that will allow us to minimize the error.

In [30]:

```
# trying to minimize error
def zero_gender(x, threshold):
    if (x[1] < threshold) and (x[2] < threshold):
        return 0
    return int(x[0])

def error_min(clf , X_test , threshold):
  y_pred_test = clf.predict (X_test)
  df = pd.DataFrame()
  df["gender"] = y_pred_test
  classes_col = [str(i) for i in clf.classes_]
  df[classes_col[0]] = None
  df[classes_col[1]] = None
  y_pred_test_proba =  clf.predict_proba(X_test)
  df[classes_col] = y_pred_test_proba
  df["new_gender"] = df.apply(lambda x: zero_gender(x, threshold), axis=1)
  return df["new_gender"]
```

In [31]:

```
# get the threshold that minimizes the error on the validation set (20% of the training set)
error = []
for threshold in range (6 , 10):
  error.append(compute_pred_score(y_val , error_min(clf, X_val , threshold/10)))

t = zip(range (6,10) , error)
best_threshold = min(t, key=operator.itemgetter(1)) [0] / 10
```

In [32]:

```
# saving the results
np.savetxt('./result/y_pred.txt', error_min(clf , X_test , best_threshold) , fmt='%d')
```

In [6]:

```
Image(url="http://robmientjes.nl/static/play/the-end.png" )
```

Out[6]: