Identify Fraud from Enron Email

Ivailo Kassamakov

May 2016

Abstract

The present project is submitted as a partial fulfilment of the requirements of the Data Analyst Nanodegree at Udacity.

All code was developed in Python, using the `sklearn` and `pandas` packages. An indispensable part is the IPython [5] notebook `p5_enron.ipynb` which contains all data preparation and algorithm training experiments. A major part of the inspiration came from the excellent [4].

Some useful tools used during the work on this project were: www.sharelatex.com and www.tablesgenerator.com.

Contents

1 Introduction **2**

1.1 Dataset structure, outliers and missing values 2

2 Creating the POI identifier **3**

2.1 Creating & Selecting features 4

2.2 Feature scaling 4

2.3 PCA decomposition 5

3 Choosing the Machine Learning algorithm **5**

4 Tuning the algorithm **5**

5 Algorithm validation **7**

6 Performance evaluation **7**

7 Conclusion **8**

8 Disclaimer **9**

List of Figures

1 Simple plot showing the values (y-axis) of the features for all records (x-axis) 2

2 Validation curve for parameter n_estimators of a RandomForestClassifier 7

3 ROC curves for the three estimators 9

List of Tables

1 Scores obtained by Logistic Regression while removing certain NaN-containing features 3

2 Features scored by the SelectKBest transformer 4

3 Scores of a Logistic Regression algorithm preceded either by a MinMaxScaler or a StandardScaler . . . 5

4 Pipeline parameters selected through GridSearchCV 7

5 Final performance scores achieved by the trained ML algorithms 8

1 Introduction

Question: Summarize for us the goal of this project and how machine learning is useful in trying to accomplish it. As part of your answer, give some background on the dataset and how it can be used to answer the project question. Were there any outliers in the data when you got it, and how did you handle those? [relevant rubric items: "data exploration", "outlier investigation"] ■

The goal of the present project is to identify Enron employees (persons-of-interest or POIs) having potentially committed a fraud. This identification is done via an analysis of a combined dataset comprising information distilled from the Enron email dataset, together with financial data.

The Enron Email Dataset (a.k.a. the Enron Corpus) is a collection of around 0.5M emails from 158 persons, most of them belonging to the Enron's senior management [2]. This dataset has been made publicly available during the Enron case investigation and is being used in a multitude of social networking, text sentiment discovery and email classification (among others) research projects around the world.

The Enron court trial resulted also in the release of detailed financial exhibits showing information about the incomes and stock values owned by many of the Enron's top management personnel.

The search for POIs in this project is performed using Machine Learning techniques as they prove rather appropriate in classifying (labeling) persons based on uncovering patterns in their behavior (in this case manifested in their financial and email relationships).

1.1 Dataset structure, outliers and missing values

The original dataset consists of 146 records (persons), each one containing 21 attributes. One of these attributes (*poi*) is the ground-truth label, the rest 20 are the features.

For easier manipulation, the input data set was transformed into a Pandas DataFrame.

A missing value analysis showed that there is a significant amount of NaN values in the dataset. There are 69 persons (i.e. 47% of all records) which have NaN values in more than half of the features. An extreme case is the record for **LOCKHART EUGENE E** that contains only N/A values.

On the other hand, all features contain at least one NaN value. The worst cases are with the *loan_advances* (has NaN values in 142 of the 146 records), *director_fees* (has NaN values in 129 records) and *restricted_stock_deferred* (has NaN values in 128 records).

Of the other features, the *email_address* is useless, as it is unique for each record and as such it has no value as a predictor for the *poi* target.

Next, a simple plot was produced to get an idea about the ranges of the numeric features. As shown in figure 1, one of the records contained values much larger than the rest. This **outlier** was identified to be the **TOTAL** record, which turned out to simply contain the sums of the remaining records.

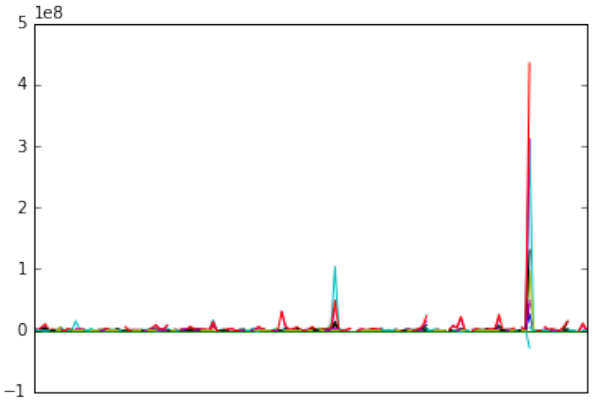

Figure 1: Simple plot showing the values (y-axis) of the features for all records (x-axis)

With the gained knowledge, a cleaned dataset was prepared, where:

- The record for "LOCKHART EUGENE E" was deleted.

- The "TOTAL" record was deleted.
- The feature "email_address" was deleted.
- The mostly-NaN-containing features "loan_advances" and "director_fees" were also removed. This lowered very little (0.3%) the *precision*, which is a good trade-off for the achieved reduction of the feature space. On the other hand "restricted_stock_deferred" was left untouched, since removing it worsened a bit both the *precision* and the *accuracy* of the estimator. Table 1 shows the performance scores achieved by the Logistic Regression algorithm when removing different sets of the said features. I've selected case 3 from the table.

Case	Features removed	Recall	Precision	Accuracy	AUROC
1	None	0.882	0.355	0.740	0.831
2	loan_advances	0.882	0.353	0.739	0.831
3	loan_advances + director_fees	0.882	0.354	0.740	0.831
4	loan_advances + director_fees + restricted_stock_deferred	0.882	0.353	0.738	0.831

Table 1: Scores obtained by Logistic Regression while removing certain NaN-containing features

Since the classifiers in the `scikit-learn` package do not accept NaN values, a *missing value imputation* was needed. Two imputation strategies were tried:

- Median—whereby the missing values are replaced with the median for each column (feature).
- Mean—whereby the missing values are replaced with the mean for each column.

Experimenting with both showed that the **median strategy** leads to better predictive power.

The two distinct classes found in the *poi* label indicate a **binary class** dataset. The distribution of these classes (126 0's vs. 18 1's for the cleaned dataset) is a sign, on the other hand, of a **highly unbalanced** dataset. This property will have repercussions on the choice of some ML algorithm parameters (see below).

2 Creating the POI identifier

Question: What features did you end up using in your POI identifier, and what selection process did you use to pick them? Did you have to do any scaling? Why or why not? As part of the assignment, you should attempt to engineer your own feature that does not come ready-made in the dataset – explain what feature you tried to make, and the rationale behind it. (You do not necessarily have to use it in the final analysis, only engineer and test it.) In your feature selection step, if you used an algorithm like a decision tree, please also give the feature importance of the features that you use, and if you used an automated feature selection function like SelectKBest, please report the feature scores and reasons for your choice of parameter values. [relevant rubric items: "create new features", "properly scale features", "intelligently select feature"] ■

A crucial step before training an ML algorithm is performing a **preprocessing** of the features. The preprocessing can include some or all of the following steps (order is important!):

Creating new features that combine or derive in a certain way from the existing ones. Commonly, sums or ratios of existing features can be good candidates for inclusion. Another option is creating non-linear combinations (i.e. higher-degree polynomials) of features. This can be useful for projecting linearly non-separable feature sets into higher-dimensionality spaces, where the ML algorithms could possibly find linear decision boundaries.

Selecting features that have the biggest importance, i.e. predictive power. An example is choosing the features having the highest variance (as the `SelectKBest` transformer does when calculating the ANOVA F-value between feature/label). Another possibility is training a Decision Tree classifier and selecting the features with the highest importance as determined by the said classifier.

Scaling, standardizing or normalizing of the features. The goal of this step is to map the features to a uniform scale, or to transform them to the same normal distribution. This can prevent for example large-value features "eclipsing" small-value ones. For many ML algorithms this a crucial requirement. SVM (especially with the "rbf" kernel), and Logistic Regression belong to this category. On the other hand, algorithms like Decision Trees or Random Forests (i.e. ones that work with decision boundaries parallel to the feature space axes) are not sensitive to the features scales, and can do without this step.

Feature space dimensionality reduction, aiming to map the existing features onto a lower-dimensionality orthonormal basis. A prominent transformation in this class is the PCA (Principle Component Analysis). PCA builds this basis by finding the directions of the highest feature variances (which are the eigenvectors of the covariance matrix). This component decomposition: 1.) reduces the following computational burden by reducing the number of effective features, and 2.) is also highly effective in eliminating the colinearity between the features. In our case the second advantage helps us avoid the manual search and elimination of "sum-of-other-features" features (like *total_payments*).

2.1 Creating & Selecting features

I approached the feature creation from two different directions:

First, several (simple) derived features were created and added to the feature set. These are:

- 'total_gains' = 'total_stock_value' + 'total_payments'
- 'from_poi_messages_ratio' = 'from_poi_to_this_person' / 'to_messages'
- 'shared_poi_messages_ratio' = 'shared_receipt_with_poi' / 'to_messages'

Later I used `SelectKBest` to select the features. I run it through a GridSearchCV iterator where different values for K were tried. The winning configuration (Logistic Regression) came out with $K = 20$ (i.e. all features were used). The ANOVA p-values calculated by the `SelectKBest` for each features in this case are given in table 2.

Interesting enough is the fact that the two new 'ratio' features have 2-3 times higher p-values that their constituents. This is not true of the new 'total_gains' features, which did not prove very useful.

Feature	ANOVA p-value
restricted_stock_deferred	0.756
from_messages	0.668
deferral_payments	0.608
to_messages	0.348
expenses	0.313
from_poi_messages_ratio	0.171
from_this_person_to_poi	0.138
other	0.049
from_poi_to_this_person	0.040
shared_poi_messages_ratio	0.011
shared_receipt_with_poi	0.007
total_payments	0.004
restricted_stock	0.004
long_term_incentive	0.004
deferred_income	0.002
salary	0.001
bonus	0.0001
total_gains	0.00009
total_stock_value	0.000003
exercised_stock_options	0.0000006

Table 2: Features scored by the SelectKBest transformer

Another algorithm (SVC) showed best results using less features ($k = 14$).

The second approach to feature creation was the use of the `PolynomialFeatures` transformer to create interaction-only combinations of features. I run GridSearch setting the *poly_degree* parameter to 1, 2 and 3. However, none of the tried ML algorithms chose any value other than 1. So, polynomial expansion of the feature space did not prove useful.

2.2 Feature scaling

I tried scaling the features with a `MinMaxScaler(feature_range=(0, 1))` and a `StandardScaler`. The MinMaxScaler gave better results. While the StandardScaler led to somewhat better accuracy in the predictions, the MinMaxScaler was better in the recall/precision scores. Table 3 shows some achieved parameters with a Logistic Regression pipeline in two experiments—one with StandardScaler, the other with MinMaxScaler.

Scaling Method	Recall	Precision	Accuracy	AUROC	PCA components
MinMaxScaler	0.882	0.354	0.740	0.831	3
StandardScaler	0.798	0.343	0.745	0.812	5

Table 3: Scores of a Logistic Regression algorithm preceded either by a MinMaxScaler or a StandardScaler

2.3 PCA decomposition

All tried algorithms were preceded by a PCA transformation. The *n_components* parameter of this step was varied inside of the GridSearch tuning. It was found that 3-4 components were enough to achieve the best predicting results.

An interesting observation is the fact that if the input dataset is first transformed with a `StandardScaler`instead of a `MinMaxScaler` , the number of PCA components used by the best case rises to more than 5.

3 Choosing the Machine Learning algorithm

Question: What algorithm did you end up using? What other one(s) did you try? How did model performance differ between algorithms? [relevant rubric item: "pick an algorithm"] ■

Since the input dataset is already labeled, we have to consider an algorithm for **supervised learning**. And since the target label (*poi*) is a discrete variable with two unique values (1 and 0, labeling effectively the person in question as being or being not a POI), we're a confronted with a **binary classification** task.

I experimented with 3 ML algorithms, pertaining to 3 different algorithm classes:

- **RandomForestClassifier**—this is an ensemble method combining (bagging) the predictions of many weaker Decision Tree estimators. I choose this one over another popular ensemble method (AdaBoost) due to its inherent parallelizability.
- **SVC**—This is a *Support Vector Machine* implementation. Selecting the right kernel is a science on its own ([1]), and since I didn't yet manage to get anything but a cursory familiarity with it, I followed some common advices ([3]) and selected the 'rbf' kernel.
- **LogisticRegression**—This is a linear algorithm, which despite its name is actually used for classification.

For each algorithm I constructed a separate **sklearn pipeline**, which combines the preprocessing as discussed above with the ML estimator in question. In particular, the steps performed by each Pipeline are:

1. A step for generating polynomial feature combinations via `PolynomialFeatures`. This step was kept during the exploratory phase, and was disabled for the final training/validation phase, since it did not prove useful (see above).
2. Select features via `SelectKBest`.
3. Perform feature scaling via `MinMaxScaler`. This scaler was chosen over the StandardScaler as shown above.
4. Perform PCA decomposition of the feature space.
5. The ML classifier itself.

Note that in compliance with `Pipeline's` requirements, the first three steps are *Transformers* (i.e. they contain a `fit_transform()` method), and the last step is an **Estimator** (i.e. it contains a `fit_predict()` or similar method).

The pipelines were fed with the 20-D feature space (the original 20 features - 3 deleted features + 3 newly created features), together with the ground truth *poi* label.

Then pipelines were separately tuned and validated. The algorithm that showed the best performance is **LogisticRegression**.

4 Tuning the algorithm

Question: What does it mean to tune the parameters of an algorithm, and what can happen if you don't do this well? How did you tune the parameters of your particular algorithm? (Some algorithms do not have parameters that you need to tune – if this is the case for the one you picked, identify and briefly explain how you would have done it

for the model that was not your final choice or a different model that does utilize parameter tuning, e.g. a decision tree classifier). [relevant rubric item: "tune the algorithm"] ■

A crucial task after selecting an algorithm is tuning its **hyperparameters**. Failing to perform well this step can result in the algorithm having an *underfitting (high-bias)* or *overfitting (high-variance)* behaviour. Neither is OK—the former will lead to poor average performance on any dataset (including the train set), the latter will lead to excellent performance on the train set and poor performance on the test set, badly scaling to different datasets.

I tuned the hyperparameters of the ML algorithms in two ways:

- Manual selection—Several experiments were manually designed and run on a each pipeline, and for each one the parameter in question was set explicitly (usually in the constructor of the Estimator).
- Selection via **Validation curves**—This approach permits determining the value of a given hyperparameter in a semi-automatic way, by observing graphically how the performance of the Estimator changes in response to the said parameter.
- Via **cross-validated parameter grid search**—I used `sklearn's GridSearchCV` to automatically run a multitude of experiments, where the values of the required parameters were automatically chosen from a parameter grid and set by the grid searcher. For each point in the parameter space the searcher runs a custom cross-validation strategy. The grid search code I used has the following generic form:

```
cv = StratifiedShuffleSplit(y, n_iter=50, test_size=0.1, random_state=42)
grid_search = GridSearchCV(clf, dict( <the parameter grid dictionary> ),
                           cv=cv, error_score=0, scoring='recall', n_jobs=-1)
```

  The reason for choosing the 'recall' scoring strategy is discussed below.

The hyperparameters that I experimented with **manually** or with the help of a **validation curve** were:

- For LogisticRegression:
  - **penalty**—Used to specify the norm for the penalization. I found the 'L2' norm to give much better performance than 'L1'. This is not surprising, giving that the 'L1' norm is used more for sparse feature spaces (which is not our case).
  - **class_weight**—Used to specify if all label classes shall have the same weight. Given that our set is highly unbalanced (much more 0 *poi* labels than 1's), it proved crucial selecting the 'balanced' value for this parameter. This drastically improved the performance.
- For RandomForestClassifier:
  - **n_estimators**—This parameter determines how many Decision Trees are to be deployed and their results combined. I determined it by producing a validation curve on the already fit pipeline (figure 2). As a result I chose a compromise value of 11.
  - **class_weight**—Similar to the case with LogisticRegression it is important to set this parameter to 'balanced' to take into account the class disbalance.
- For SVC:
  - **class_weight**—Similar to above, this parameter has to be set to 'balanced' to get any useful performance.

The hyperparameters that were selected via **GridSearchCV** were:

- For all pipelines:
  - **SelectKBest's K**—This is the desired number of features to be selected.
  - **PCA's n_components**—This is the desired number of principal components.
- For LogisticRegression:
  - **C**—The inverse of the regularization strength. Higher values mean lower regularization, i.e. more weight coefficients are different from 0.
- For RandomForestClassifier:
  - **min_samples_split**—This is the minimum number of samples required to split an internal node.

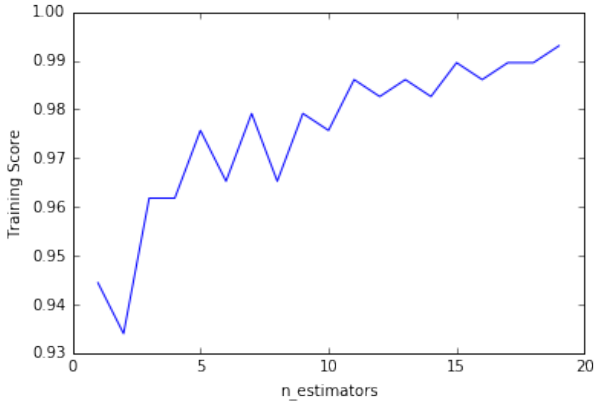

Figure 2: Validation curve for parameter n_estimators of a RandomForestClassifier

  - **min_samples_leaf**—This is the minimum number of samples in newly created leaves.

- For SVC:
  - **C**—This is the penalty parameter of the error term. Low C creates a smoother decision surface at the expense of higher missclassification rate. Higher C values mean more samples are selected to become support vectors.
  - γ—This is the kernel parameter for the 'rbf' kernel.

The optimal hyperparameters, i.e. the ones found by the GridSearch tuner to give best performance to the classifiers, are shown in table 4.

Estimator Pipeline	K	n_components	Param1	Value1	Param2	Value2
LogisticRegression	20	3	C	0.0001	n/a	n/a
RandomForest	18	3	min_samples_split	2	min_samples_leaf	4
SVM-rbf	14	4	C	0.1	γ	'auto'

Table 4: Pipeline parameters selected through GridSearchCV

5 Algorithm validation

Question: What is validation, and what's a classic mistake you can make if you do it wrong? How did you validate your analysis? [relevant rubric item: "validation strategy"] ■

Validating an algorithm means determining its performance in a veritable way. A classic mistake would be obtaining the performance on the same dataset that has been used for the training. Therefore a certain protocol needs to be followed during the validation. Normally it consists of splitting the initial dataset into separate subsets used for training and testing. This approach is termed *cross-validation*.

I performed the algorithm validation using the `StratifiedShuffleSplit` cross-validation iterator, whereby the dataset is split 1000 times into shuffled train-test subsets that preserve the percentage of samples for each class (important for unbalanced datasets!). The number of splits and the random_state of the iterator are chosen to correspond to the similar parameters used in the grading script `tester.py`.

This cross-validation iterator is fed to the `cross_val_score()` method that calculates different validation scores.

6 Performance evaluation

Question: Give at least 2 evaluation metrics and your average performance for each of them. Explain an interpretation of your metrics that says something human-understandable about your algorithm's performance. [relevant rubric item: "usage of evaluation metrics"] ■

Selecting the right performance metrics is crucial for the correct tuning of the ML algorithm, and depends to a great extent on the class properties of the dataset as well as our application needs.

For example if we were to use the *accuracy score* metrics we would get a rather incorrect idea about the performance of our algorithm. The reason for this is that the dataset is highly unbalanced, and even if we were to simply guess

a POI class of 0 for each sample we would achieve a stunning $126/(18+126) = 0.875$ accuracy. The *accuracy* shows how many samples have been correctly labeled among all sample population, and is defined as:

$$accuracy = \frac{tpr}{tpr + fnr + tnr + fpr}$$

In the above formula, *tpr* is the proportion of identified *true positives*, likewise *tnr* is the *true negative rate*. The *fpr* acronym denotes the *false positive rate*, also known as the *false alarms, or Type I errors*. Likewise, *fnr* stands for the *false negative rate*, which is the number of *misses, or Type II errors*.

In our application, however, we are more interested in correctly labeling only the samples belonging to the POI=1 class (this is our *positive class*). That is, we would like to be able to identify as many of the POIs as possible even taking the risk that we identify as a POI someone who is not. The latter case just increases the burden of investigating a possibly innocent person, but we will be pretty certain that we catch all the true culprits. In other words, we would like to have a high *tpr* while keeping a low *fnr*. All this translates to the requirement that we have to **maximize the recall (sensitivity)** of our algorithm. The *recall* is defined mathematically as:

$$recall = \frac{tpr}{tpr + fnr}$$

Although we generally don't mind identifying innocent people as POIs (meaning we have a certain amount of false alarms or type I errors), we would still be glad if we keep this limited. This means keeping the *precision score* above a certain limit, which follows from the definition of *precision*:

$$precision = \frac{tpr}{tpr + fpr}$$

The requirement for this project is $precision > 0.3$.

In many cases there is a trade-off (inverse relationship) between the recall and precision, therefore it could be useful comparing the performance using the F_1-score which is a combination (actually, the harmonic mean) of recall and precision, i.e.:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

Higher F_1-score means relatively better performance (i.e. both recall and precision are high).

Another useful metrics is the AUROC (area-under-ROC curve). This number essentially shows the probability that the classifier will rank a randomly chosen positive sample higher than a randomly chosen negative one.

To optimize the ML algorithms for high recall, I've set the parameter **'scoring' = 'recall'** on all cross-validation iterators.

After training the ML algorithms I performed a cross-validation scoring similar to the one used in the `tester.py` script (1000 stratified shuffled splits into 90/10 train/test subsets) and achieved the performance metrics shown in table 5.

Algorithm	Confusion Matrix	Recall	Precision	F_1 score	Accuracy	AUROC	Training time
LogisticRegression	$\begin{pmatrix} 16 & 2 \\ 35 & 91 \end{pmatrix}$	0.882	0.354	0.493	0.740	0.83	22 s
RandomForest	$\begin{pmatrix} 15 & 3 \\ 17 & 109 \end{pmatrix}$	0.500	0.354	0.389	0.804	0.79	4 min
SVM-rbf	$\begin{pmatrix} 16 & 2 \\ 32 & 94 \end{pmatrix}$	0.817	0.333	0.461	0.731	0.77	41 s

Table 5: Final performance scores achieved by the trained ML algorithms

It is easily seen that the best performance (highest recall, and also highest F_1-score and AUROC!) is achieved by the **LogisticRegression** algorithm. At the same time all trained estimators achieve the base precision score of 0.3. We also see a big difference in training times with the RandomForest algorithm taking almost 4 min!

The ROC (Receiver Operating Characteristic) curves, showing the performance of the classifier when its decision threshold is varied, are shown in figure 3.

7 Conclusion

I selected and trained three estimators (LogisticRegression, RandomForest and SVC) in an attempt to learn them to classify Enron personnel suspected of fraud. The estimators worked on a class-unbalanced pre-labeled dataset of a relatively modest size (144 samples).

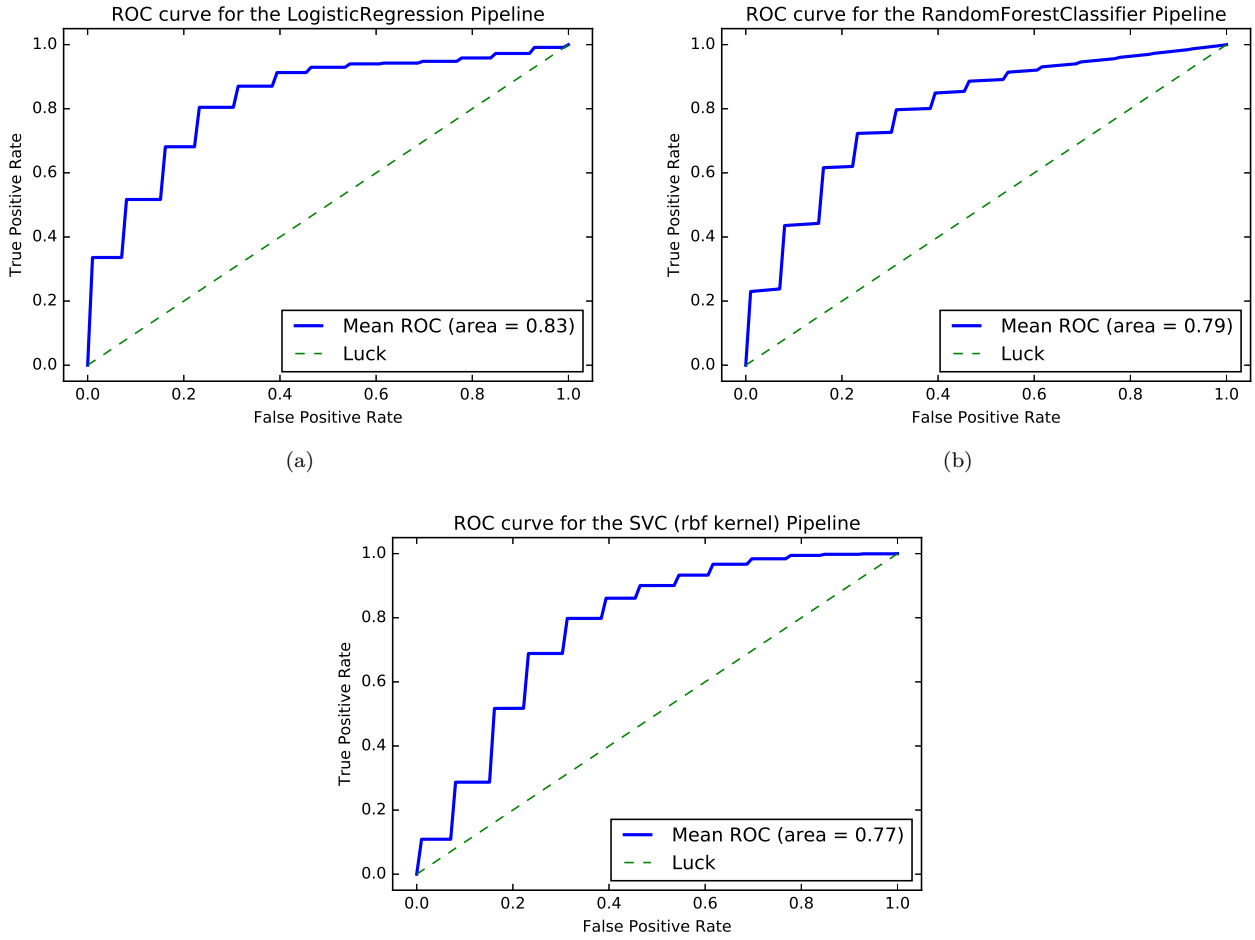

Figure 3: ROC curves for the three estimators

The highest recall score of 0.882 was achieved by the LogisticRegression, however I still consider it unsatisfying. I would prefer seeing a score greater than 95%. I see a possible way of improving the performance in experimenting with more new synthetic features. Still, I'm afraid that achieving better performance on such a small set could mean the estimator is overfitting.

I would also like to try selecting features with a Decision Tree (e.g. via ExtraTreesClassifier) and better study the effects of applying a KernelPCA instead of linear one. Why not trying other estimators like Ridge or LinearSVC too?

Another area I wished I could spend more time on is analyzing the RandomForestClassifier for overfitting (for which I have strong suspicion), as well as plotting its decision graph.

All in all, this course and project provided an excellent opportunity to gain basic knowledge and skills in ML in general, and in the Python ML packages in particular. It won't be exaggerated to say that a door to an enormous and extremely exciting area was opened to me.

8 Disclaimer

I hereby confirm that this submission is my work. I have cited below the origins of any parts of the submission that were taken from Websites, books, forums, blog posts, github repositories, etc.

References

[1] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. "Kernel methods in machine learning". In: *Ann. Statist.* 36.3 (June 2008), pp. 1171–1220. DOI: 10.1214/009053607000000677. URL: http://dx.doi.org/10.1214/009053607000000677.

[2] B. Klimt and Y. Yang. *Introducing the Enron Corpus*. Mar. 30, 2016. URL: http://ceas.cc/2004/168.pdf.

[3] Charles H. Martin. *How does one decide on which kernel to choose for an SVM (RBF vs linear vs poly kernel)?* Apr. 9, 2012. URL: https://www.quora.com/How-does-one-decide-on-which-kernel-to-choose-for-an-SVM-RBF-vs-linear-vs-poly-kernel.

[4] Sebastian Raschka. *Python machine learning : unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Birmingham, UK: Packt Publishing, 2015. ISBN: 1783555130.

[5] Cyrille Rossant. *Learning IPython for interactive computing and data visualization*. Birmingham, UK: Packt Publishing, 2013. ISBN: 9781782169932.