Udacity-DAND-Project5-PeidaCai

Approach to project

The bulk of work was done using jupyter notebook, including EDA and model testing with gridsearch and different algorithms. I felt more comfortable with using the notebook especially being able to visualize plots together with the code as well as jotting down comments on the same notebook using Markdown.

I have attached the notebook in the zipped file, but the poi_id.py file only contains the chosen classifier. Results of the trials were summarized in this pdf file below.

Data Exploration

There were a total of 146 rows in the data, and only about 12% of the data were POIs. This posed a 2-fold problem:

Small dataset - validation problem.
Firstly, 146 samples should be considered a very small dataset. There would be issues with validation (to be elaborated further later).

Class imbalance.
Secondly, since this was a binary class problem (POI vs non-POI), with only 12% of POI, there was an imbalance in the distribution of POIs. This may cause problems in modeling later, specifically in the selection of model performance metric. Accuracy should not be used since the baseline accuracy, by calling all non-pois, is already at about 88%. However, such a model would not be able to catch any POIs since every input would be classified as a non-POI.

Features used.
There were 19 different features (excluding “POI”), however, 6 of these features had over 50% of missing data, even with a robust imputation strategy, these features would contained over halve of “guessed” values, which would impact the ML model and hence its performance. (Refer to poi_id.py for specific features used)

Imputation strategy.
Therefore, I only used the remaining 13 features with less than 50% missing values and used zeros for all missing values, after looking through the pdf financial document and deemed that missing values likely were the results of that particular individual not having that specific segment of compensation (i.e. no loan-advances).

Outlier management

Total of 4 outliers were identified:

1. “TOTAL”
    a. When salary histogram was plotted, it was clear there was a single entry which clearly had an extreme value. Once it was isolated, it turned out to be the “TOTAL” row, which was removed.

2. 'THE TRAVEL AGENCY IN THE PARK'
    a. From the pdf financial document, there was another non-personnel entry, 'THE TRAVEL AGENCY IN THE PARK', which was also deleted.

3. 'BELFER ROBERT' and 'BHATNAGAR SANJAY'
    a. Since there were the features “total_payments” and “total_stock_value”, summing their respective components and entries which didn’t tally were removed.

New Features

Expense-Salary-ratio.

Expense salary ratio was created as a new feature as we would expect typical expenses to be less than salary. For an average individual, expenses should be a low, reasonable amount, compared to salary. For POIs, there may be an increased likelihood that they might have charged an astronomical amount, since expenses may be harder to account for than salary.

Expenses by itself may not be able to predict this properly since some executive may have to entertain clients and hence may end up with larger expense account. However, we should expect the expenses to commensurate their salaries. A high performer (higher salary) bringing in large account may be entitled to expense more.

POI email ratios.

Additional email features were added. Instead of just using the absolute number of emails to and from POIs, ratios of POI emails over total to and from emails were used (e.g. Emails from POI / Total email received).

1. 'To_poi_ratio'
2. 'From_poi_ratio'
3. 'shared_poi_ratio'

An individual may simply be sending (or receiving) a lot of emails, including a large number to POIs, however, proportionately, the emails to POI account for a comparatively low proportion of the total emails he sent. Therefore, POI email ratios should be a better gauge of the relationship between an individual and the POIs.

Feature selection.

SelectKBest with gridsearch (searching from minimum of 5 features to the entire set) was used to select the best features. Turned out that newly created features didn't improve the f1 scores of the models used, but whenever the new features were included, all the email ratios were almost always chosen.

Boxplots were plotted in the jupyter notebook after the new features were created to observe the distribution between non-POI and POIs, the email ratios boxplots indicated better discriminatory abilities compared to the expense-salary ratio, hence the SelectKBest results were expected.

Models attempted and scores

Modeling conclusion

Modeling was conducted in the following manner:

1. Compare plain vanilla model on all original features with less than 50% missing values.
2. Select best performing model (GaussianNB and Decision Tree) conduct GridsearchCV on serialized transformations:
3. a. StandardScaling b. SelectKBest c. PCA
4. Conduct GridSearchCV on parallelized transformation:
5. a. Standard or MinMax Scaling b1. SelectKBest on all original features b2. PCA on all original features
6. Repeat modeling process with new features (self-created ones)
7. Last approach tried was to separate create principal components for finance and email features and add these components to PCA conducted on all features and features selected using SelectKBest.
8. Models were compared based on f1 score using 100 folds stratifiedshufflesplit on the entire dataset. This was due to the small size of the data and the imbalance in the POI distribution, hence it would not be meaningful to use the traditional train/test/validation split or K-fold cross validation.

The best performing model was:

Base Model	**Number of features selected (All = features with < 50% missing values)**	**New features?**	**Scaling**	**FeatureUnion / Pipeline**	**PCA? (no. of components)**	**f1-score**
Vanilla GaussianNB	All	No	Yes	Pipe	No	0.257
Vanilla DecisionTree	All	No	Yes	Pipe	No	0.244
Adaboost (GaussianNB)	All	No	Yes	Pipe	No	0.181
Adaboost(LogReg	All	No	Yes	Pipe	No	0.175
GaussianNB	10	No	Yes	Pipe	4	0.320
GaussianNB	7	No	Yes	Pipe	No	0.305
GaussianNB	5	No	No	Union	4	0.326
GaussianNB	7	No	Yes	Union	3	0.335
GaussianNB	5	No	Yes	Union	2	0.340
GaussianNB	7	Yes	Yes	Union	2	0.317

GaussianNB with MinMaxScaling and parallelized SelectKbest and PCA on all original features with less than 50% missing values.

Selected features.
'bonus', 'exercised_stock_options', 'salary', 'total_payments', 'total_stock_value'

What is model tuning?

Every algorithm or model has some arguments which can be varied for different performance values.

In this case, GaussianNB was chosen with SelectKbest and PCA. Parameters chosen for tuning were the number of components used from PCA and the number of features selected using SelectKBest.

Model tuning means changing these parameters (via GridSearchCV or manually) and comparing the results after each change. Results here was chosen as f1 score instead of accuracy. Accuracy is not a good metric in this case since there are a strong imbalance in the classes, 88% non-poi compared with 12% POI. Simply by calling all test data a non-poi, the "model" would achieve a good accuracy score of about 88%, but failed in its intended use since it can never identify any POI.

F1 score takes the harmonic mean of precision and recall scores, and provides a more relevant metric to measure the performance of the models in this case.

Recall is a measure of the model's ability to identify the POIs in the test set. It is the ratio of the number of correctly identified POIs over all POIs in the test set. If say, there were 5 POIs in the test set and the model managed to identify 2, the recall is 0.4.

Precision is the ratio of correctly identified POIs over sum of correctly identified POIs and incorrectly identified POIs. This serves as a kind of check to avoid the other extreme of calling all test data POI, which would achieve 1.0 recall but ends up with large number of wrongly identified non-POIs (calling a non-POI a POI). In the real world, such error would result in higher unnecessary work, such as in this case, the human investigator would have to interview a lot more non-POIs who were wrongly classified as POIs.

Model tuning is important as it allows for model to be “customize” via their parameters to this particular dataset to maximise performance. In the case of gradient descent, without trying other values for parameters or changing other parameters, the model may be optimized for a local minimal (of the loss function), while there may exist a lower (better) global minimal which could be found via tuning. Having said that, computational resources should also be taken into consideration. In cases where the dataset is large, running large permutation of parameters into the model may yield small percentages of performance increment but may be done at the expense of time (could be a matter of days or weeks of gridsearching).

What is validation and why is it important?

In ML, the traditional approach is to break up the dataset into train, test and validation set (about 0.7, 0.2, 0.1, but really depends on the characteristics of the dataset). The purpose is to hold out data when training the ML model so as to avoid overfitting.

The general process is to use the training set to fit a model, use the test set, which at this point was never seen by the model, to check the performance of the model. Re-tune the model if necessary, then use the validation set to check the performance of the model again.

Since we can never get a dataset which encompasses all possible scenarios, trying to tune a model to generate best performance using all the data may result in over-fitting to only the training data, and the model being unable to generalize well to unseen data, hence resulting in poor real-world performance.

In the event when the dataset is huge, separating it into the 3 sets in a random manner would be feasible. However, when the dataset is small, we may need to use cross-validation (usually k-fold cross-validation), where the dataset may be separated into 2 (train/test set and validation set). Within the train/test set, the data is sliced into a number of smaller set (k-fold), say 5. The cross-validation process then holds out 1 of the 5 sets as a test set, train the model with the remaining 4 datasets and use the 1 to test the model. The process repeats this method for k (5 in this case) times so that each of the k set "gets a turn" to be the test set. The performance metric is then averaged over k times to provide an average score.

Then there is this case of even smaller dataset with imbalanced classes. Using k-fold cross-validation may not be possible since it is likely the folds end up without some or 1 class of the data. I.e. the model was trained only on non-POI data. In this case, we had to use stratifiedshuffle split and give up the concept of a validation dataset, since holding out on precious POIs for validation means the training process loses significant portion of "learning materials" on the POIs, hence resulting in an over-generalized model.

Therefore, in this case, the entire dataset was used for the stratifiedshufflesplit. Stratifiedshufflesplit was used also because the process ensures that the classes are always split in the required manner. In my model, I used a 0.7/0.3 split and 100 folds. This means that each time the data is split to train and test set, there would be 30% of the POIs in the test set and 70% of the POIs in the training set. However, this process doesn't guarantee that each split is unique. This is overcame by the 100 folds (or possibly more folds, if computational power is available).

It is worthy to note that once the dataset get large, cross-validation (k-fold or stratified) becomes a computationally expensive affair and the marginal increment in model performance may not be justified.