Accomplishment Classifier with Machine Learning

Kalia Barkai

Minerva Schools at KGI
San Francisco, CA 94103
kalia@minerva.kgi.edu

August 2017

The Python classifier model categorises the accomplishments submitted by Minerva applicants into 1 of 50 different categories. The classifier is built by vectorizing the accomplishments with Tfidf and then is fitted on a Logistic Regression on 75% of the accomplishments and their labels and tested on the remaining 25%. Over- and under-sampling methods are used due to the imbalanced nature of the data. To analyse the data and results the following visualisation methods are used: learning curves, ROC (Receiver Operating Characteristic) and LIME (Local Interpretable Model-Agnostics Explanation). The final accuracy of the classifier falls around 0.58 to 0.62.

1 Introduction

Minerva's application includes a number of different challenges which judge whether or not the student would be a fit for the university. One of these challenges is the accomplishment section where the applicant has the opportunity to describe up to 5 accomplishments. These can be anything that the applicant is proud of and would like to share with Minerva. Each accomplishment has two parts which are a maximum of 500 characters each: "description" and "description of impact".

2 Background

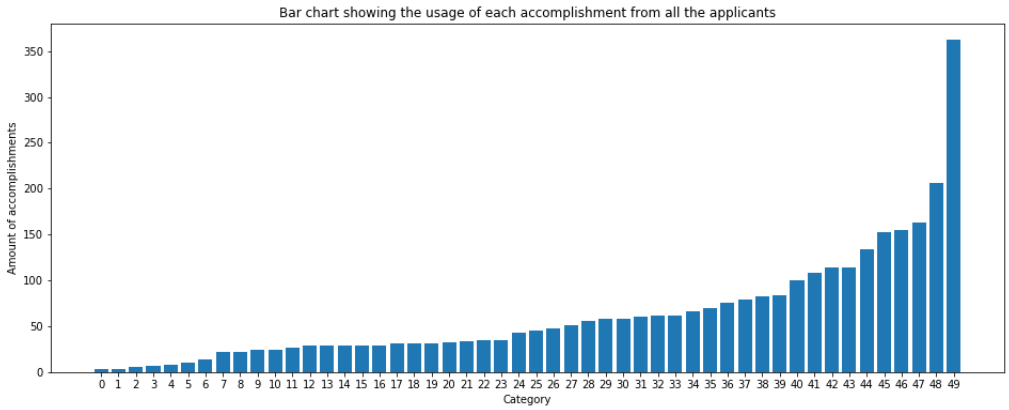
Application processors go through an applicant's accomplishments and categorize each one based on 50 different categories which fall under 9 broader headings.

This classifier aims to automate this system by using natural language processing and machine learning to categorise accomplishments by finding a pattern between the words used in the text and the previous ways these accomplishments have been classified.

3 Approach and Experiments

The algorithm reads in a csv, using Pandas, with the following column headings: "category", "description", "description_of_impact", "mid", "grader" and "accpk", where "mid" is the Minerva ID of the applicant, "grader" is the ID of the application processor and "accpk" is the ID of the accomplishment. The text which will be used in the classifier is the combination of "description" and "description_of_impact" for each accomplishment as it was found that combining these two bodies of text per accomplishment resulted in higher accuracy for the classifier.

To get an understanding of the distribution of the data, Bar Chart 1. below shows the frequency of each category used by all the applicants. Here it is possible to see that the data is greatly imbalanced. If a classifier trains on imbalanced data, then the classifier will see the pattern of there being a greater probability that an accomplishment will fall into one of the majority classes. And so, to increase accuracy, the classifier will be biased to classify more accomplishments in majority classes, because, chances are, regardless of their content, that they are one of these categories. Therefore, it is necessary to tune the classifier, so that it takes into account the imbalanced nature of the data being fitted to it.



Bar Chart 1. Bar chart showing the amount of accomplishment per category, labeled here from 0 to 49. Category 21 has 362 accomplishments while categories 0 and 1 only have 3 accomplishments each.

3.1 Cleaning Text and Dividing Data

To clean each accomplishment the Natural Language Toolkit (NLTK) package is used. During this process punctuation is removed and each word is reduced to its stem. The stem of the word is its most basic form that is still unique to the topic of the word. For example the words "debating" and "debate" would both be reduced to "debat" and since "debating" and "debate" serve the same purpose in being a possible category for the whole text, it makes more sense to combine their influence on the classifier by making them the same word. Listing 1. below, shows the code used to clean the accomplishments.

```
# import stemmer to reduce words to stem words only
from nltk.stem.snowball import SnowballStemmer
# import string to remove punctuation from strings
import string

# function to clean text
def cleaner(content):
    words = ""
    # remove punctuation
    text_string = content.translate(string.maketrans("", ""),
   string.punctuation)

    # split words
    to_stem = text_string.lower().split()

    #stem words
    x = 0
    stemmer = SnowballStemmer("english")
    while x < len(to_stem):
        words += " " + stemmer.stem(to_stem[x]) + " "
        x +=1

    return words
```

```
23
24 # clean all text in features
25 x = 0
26 features_clean = []
27 while x < len(features):
28     clean = cleaner(features[x])
29     features_clean.append(clean)
30     x += 1
```

Listing 1: Code used to remove punctuation and stem words in each accomplishment where "features" is the list of uncleaned accomplishments.

To be able to test the classifier, the data needs to be separated into a training set which is fitted to the classifier and a testing set. The classifier makes its observations based on the patterns in the training set, and then tests how it classifies on the testing set. Using the train_test_split model selection tool from sklearn, the data is separated into a training set which consists of %75 of the data, and therefore a testing set of %25 of the data. This can be seen in Listing 2.

```
1 from sklearn import model_selection
2
3 features_train, features_test, labels_train, labels_test =
    model_selection.train_test_split(features_clean, labels)
```

Listing 2: train_test_split model selection tool code where features_clean is the data that has been cleaned through the cleaner function defined in Listing 1.

3.2 Tfidf Vectoriser

Once the accomplishments have been cleaned, it is then necessary to vectorise each accomplishment by transforming it into a numerical array of features used to train and test the classifier. This means that each stem word throughout all the text will be given a unique number, this is called **tokenising**. For example, take into consideration the following text:

Every weekend I go to the beach. Last weekend, I took some tourists with me. (1)

In the above text, “every” would be assigned the number 0, “weekend” would be assigned the number 1 and so on. However, in the second sentence, when “weekend” is written again, it will not be assigned a new number, but will be counted as another use of word 1 in the text.

Each accomplishment will then be assigned the unique number of all the words which appear in its body of text and the amount of times it appears (**counting**). And finally, based on the the frequency of each token throughout all the accomplishments, the importance of a token is reduced, if it appears in the majority of accomplishments, or increased, if it is rarer, by assigning it a weighting, and this is called **normalising**.

Each token is known as a **feature** and each accomplishment as a **document**. For example, if we look at text (1) above, the two sentences would be one document, and word 1, which is “weekend”, and its frequency in the document, 2, is a feature.

The Tfidf vectoriser from the sklearn package extracts the features from the accomplishments in this way and in doing so transforms them into a **Bag of Words** which is one of the supported representations for features in classifier algorithms. Tfidf vectoriser used in this case had two parameters which were checked or adjusted;“stop_words” and “ngram_range”.

If “stop_words” is set to a list (such as “english”) it removes all the words from the accomplishment within that list, such as “and”, “or” and other conjunctions and articles which do not refer to a specific subject.

“ngram_range” chooses how many words together to tokenise. For example, if ngram_range = (1,1) then a single word will be one feature and this is known as a 1-gram. If the ngram_range is set to tokenise 2-grams, then each feature will be two words, which appear next to each other in the text. For example, in text (1), “Every” would be a 1-gram token, but “Every weekend” would be a 2-gram token. If ngram_range = (1,2) then the vectoriser will produce both 1-grams and 2-grams. For this parameter, the first value always refers to the minimum-gram and the last value to the maximum-gram.

GridSearchCV from sklearn, which is a model selection tool, was used to test a number of different settings for each parameter in the vectoriser and classifier, by trying out a given set of parameters and their combinations. The data which each combination is tested on uses a StratifiedKFolds method to make the splits on the main dataset. A fold refers to a subsample of the dataset. This means that the dataset is split into k folds to be tested on, where the k-1 remaining folds are used to train the classifier. With StratifiedKFolds, the percentage of

samples per class is the same in each fold. The default number of splits, and therefore the value for k, is 3. GridSearchCV was therefore fitted using all the data.

From this, the vectoriser used in this code used an ngram range of (1,1) which gave the best results. Listing 3. shows how the parameters were defined in the code

```
from sklearn.feature_extraction.text import TfidfVectorizer

# vectorize words and frequency in each string in "features"
   while removing stop words (e.g. a, the, ...)
vectorizer = TfidfVectorizer(stop_words= 'english', ngram_range =
    (1,1))
```

Listing 3: Code used to import and define the Tfidf vectoriser from sklearn.

3.3 Random Forest Classifier

Random Forest is a type of ensemble classifier. This means that it uses a number of Decision Trees on samples of the accomplishments and then averages the results from all the Decision Trees to create one algorithm. The advantage of using an ensemble method is that since each Decision Tree might have a slightly different error, by averaging their results, the errors will not be passed on to the final algorithm. Also, since Random Forests are built from a number of Decision Trees, it is also meant to mitigate overfitting, as the algorithm makes decisions based on a number of different patterns used on the training set, and not just one "optimal" pattern for that training set. Decision Trees work by finding features which help the algorithm make a decision on which class the accomplishment falls under. It creates a number of these decisions, starting with the features it thinks create the biggest split between classes and following the decisions until it reaches a category. This is explained in Figure 1. below.

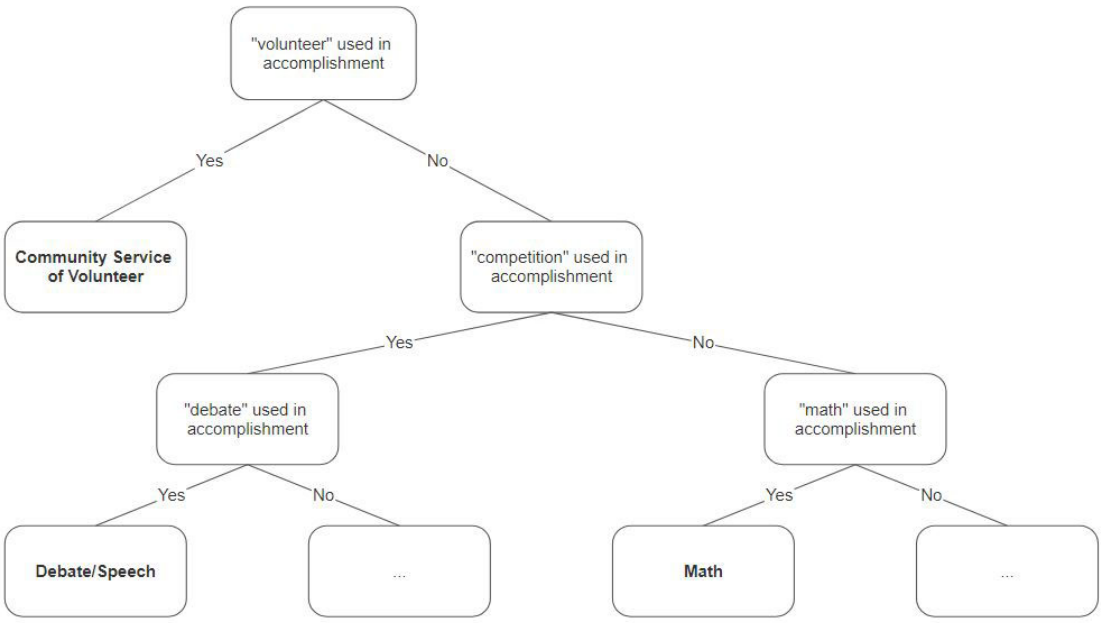


Figure 1. Example of simplified decision tree learned during classifier fitting. This decision tree looks for a certain word in the accomplishment to be able to make a final classification.

By using GridSearchCV the following parameters were checked and/or adjusted in the Random Forest classifier: "n_estimators" and "max_features", where "n_estimators" is the number of Decision Trees and "max_features" is the number of features to consider when choosing between which categories to classify the accomplishment in. The vectoriser and classifier are then run through a pipeline which takes the training data transforms it using the vectoriser and fits it to the classifier. This can be seen in Listing 4. below:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import make_pipeline

clf = RandomForestClassifier(n_estimators = 400, max_features = "
    auto")

forest = make_pipeline(vectorizer, clf)

forest.fit(features_train, labels_train)
predictions = forest.predict(features_test)
print forest.score(features_test, labels_test)
```

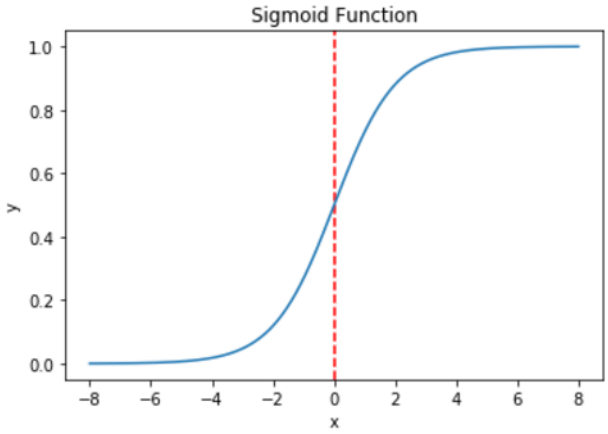
```
>> 0.546835443038
```

Listing 4: Random Forest classifier "clf" code and pipeline "forest" used to transform and fit training data. "predictions" are the classes assigned to the untrained features from this classifier. The result of running this classifier this time was around 0.55 accuracy.

3.4 Logistic Regression

The next classifier tested on the data is Logistic Regression. This classifier is used to make binary decisions. For example, if we were just deciding if the accomplishment was "Music: Vocal" versus "Music: Instrumental" and could only be one or the other, in this case "Music: Vocal" could be 1 and "Music: Instrumental" could be 0. Logistic Regression relies on the Sigmoid function (Equation 1.) as a base for its Hypothesis function (Equation 2.).

$$sig(t) = \frac{1}{1 + e^{-t}} \tag{1}$$



Graph 1. Graph representation of the Sigmoid function.

$$h_\theta(x) = sig(\theta^T x) \tag{2}$$

Here, x is the input feature(s) and θ is the parameter(s) which is defined from the training data fitted to the classifier. $\theta^T x$ creates a function of a graph, where according to the position of the data points on the graph, we can see if they fall within the classification or not. For example, if the function creates a straight line, then we might know that data points below this line are in Class 0 while data points above it are in Class 1.

The hypothesis function then results in the probability that the input is in Class 1, which

in our example is "Music: Vocal". This can be seen from the result which will be a number between 0.0 and 1.0, where the closer it is to 1.0 the more likely that the input falls in Class 1. From here, the classifier uses a threshold (default 0.5) to show that any probability less than 0.5 classifies the input as 0 and any probability greater than or equal to 0.5 classifies it as 1.

However, since this the data used rely on a multi-class solution, it is not enough for the classifier to make binary decisions. Logistic Regression from sklearn has a built in parameter "multi_class" which can be set to "ovr" (which stands for "one versus rest") and allows the classifier to make a binary decision per each class. Additionally, the classifier can be built into a multi-class tool, also from sklearn, called OneVsRestClassifier which creates a different classifier for each class, such that the class is equal to 1, and anything not that class is equal to 0.

Logistic Regression from sklearn has another useful parameter called "class_weight", which, if set to "balanced" gives a weighting which influences the θ parameter depending on the frequency of that class within the training data. Since the data we have is very imbalanced, this parameter increases the accuracy of the classifier greatly. Listing 5. below shows the code used to implement the Logistic Regression classifier, again the vectoriser and classifier are run through a pipeline.

```
1 from sklearn.linear_model import LogisticRegression
2 from sklearn.multiclass import OneVsRestClassifier
3 from sklearn.pipeline import make_pipeline
4
5 classif = OneVsRestClassifier(LogisticRegression(class_weight = '
    balanced'))
6 c = make_pipeline(vectorizer, classif)
```

Listing 5: Code showing implementation of Logistic Regression and OneVsRestClassifier

3.5 Feature Selection

The last component added to this pipeline is a dimensionality reduction tool from sklearn called TruncatedSVD. This algorithm reduced the amount of features which the classifier uses to make its decisions from. This is useful when the reduced features are those which hold the most influence on whether or not an accomplishment falls within a specific category. In a way, the dimensionality reduction allows the classifier to make more informed decisions. It does this

by evaluating the features, for example if two features serve the same or very similar purposes then the second feature does not contribute any new information and is therefore redundant. The SVD reduces those features to zero and then this transformed set of training data is fitted to the classifier. Listing 6. shows the code which defines the TruncatedSVD used in this classifier. The parameter "n_components" is the amount of features to keep, and in this case it is chosen to be similar to the amount of training data as this results in the highest accuracy scores.

```
from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=3000, n_iter=7, random_state=42)
```

Listing 6: TruncatedSVD feature selection tool

Listing 7. shows the pipeline with the vectoriser, SVD and classifier and the resulting accuracy after fitting the training data.

```
c = make_pipeline(vectorizer, svd, classif)
c.fit(features_train, labels_train)
predicted = c.predict(features_test)
print c.score(features_test, labels_test)
>>0.587341772152
```

Listing 7: Full pipeline and implementation of the vectoriser SVD and Logistic Regression classifier. The accuracy score of the classifier this time is around 0.59.

However, this feature selection method does not actually improve the accuracy of the classifier because Logistic Regression supports regularisation by default. This means that the algorithm already tries to reduce "noise" in the features by increasing the weight, and therefore the influence, of features that help differentiate between categories, and reducing the weight of those that do not.

3.6 Over- and UnderSampling Methods with imblearn library

Resampling is the act of adjusting the main dataset so as to try balance imbalanced data. This can be done by either oversampling minority classes, undersampling majority classes, or both.

3.6.1 Oversampling with SMOTE

SMOTE stands for Synthetic Minority Oversampling Technique. It is an algorithm which takes the classes with less training data and creates more examples of these classes from a combination of their prominent features. SMOTE from imblearn, has a parameter which allows it to use it's own oversampling ratio or for the user to input the exact ratios they would like to change. The SMOTE algorithm cannot create more examples for classes which have fewer than 6 examples already in the training set, therefore it is necessary to input manually the new ratio for only those minority examples over 6. This was done by increasing the number of examples depending on the interval the initial number fell in. The ratios chosen were a matter of trial and error, and can be seen in Listing 3.

It is important to only apply an oversampling method to the training data because if it would be applied to all the data, then the testing data will have examples very similar to those in the training set and this would result in a high accuracy score, but would not be generalisable to new data.

Listing 8. shows the implementation of SMOTE with the Logistic Regression classifier.

```
1 from collections import Counter
2 print Counter(labels_train) # checks current category ratio
3 >> Counter({22:49, 5:48, 6:45, 36:44, 47:43, 44:43, 49:40, 13:39,
     3:37, 12:36, 32:35, 4:33, 31:30, 41:27, 7:26, 14:26, 25:25,
    42:24, 9:23, 35:23, 1:23, 30:19, 43: 19, 20:16, 45:15, 34:13,
    26:10, 39:9, 2:7, 37:6}})
4
5 # the numbers on the left of the ratio are the class number,
    while the numbers on the right are the new example amounts for
     that class.
6 sm = SMOTE(ratio = {22:50, 5:50, 6:50, 36:50, 47:50, 44:50,
    49:50, 13:50, 3:50, 12:50, 32:50, 4:50, 31:50, 41:50, 7:50,
    14:50, 25:50, 42:50, 9:50, 35:50, 1:50, 30:40, 43: 40, 20:40,
    45:40, 34:40, 26:40, 39:30, 2:30, 37:30},          kind='
    regular', random_state=42)
7
```

```
# need vectorized features to oversample
features_trained = vectorizer.fit_transform(features_train)
# resampled data
features_res,labels_res = sm.fit_sample(features_trained.toarray
    (), labels_train)
# fit and transform on svd
features_train_2 = svd.fit_transform(features_res)
# fit on OneVsRestClassifier of Logistic Regression
classif.fit(features_train_2, labels_res)
features_testing = vectorizer.transform(features_test)
features_test_2 = svd.transform(features_testing)
predict = classif.predict(features_test_2)
print classif.score(features_test_2, labels_test)
>> 0.616455696203
```

Listing 8: Classifier refitted with resampled data from SMOTE.

3.6.2 Undersampling Methods

Undersampling is a another method used to mitigate the effect of imbalanced data. Undersampling algorithms take the majority classes and reduce the number of examples in the training set to try equalise the distribution of all categories in the training data. Again, it is advised to use undersampling on the training set, as there is no reason to reduce the testing set, which will just give you less examples to test the classifier on.

Eight different undersampling methods were tested and their accuracies are compared in Table 1.

Undersampling method	Accuracy Score
TomekLinks	0.576
CondensedNearestNeighbour	0.275
EditedNearestNeighbour	0.367
RepeatedEditedNearestNeighbour	0.367
AllKNearestNeighbours	0.513
NeighbourhoodCleaningRule	0.476
OneSidedSelection	0.352
RandomUnderSampler	0.619

Table 1. Table showing the different undersampling methods tested and their respective accuracy scores on the same training data.

From Table 1. we can see that TomekLinks and RandomUnderSampler performed the best. One difference in the implementation of these two undersamplers is that for the RandomUnderSampler the ratios can be set by the user, and this was the case here. TomekLinks works by removing the majority class's Tomek Links from the sample. These are the examples where the nearest neighbour for a given data point is from a different class. RandomUnderSampler, on the other hand, randomly removes samples from the majority classes.

4 Analysis

From the different methods explored in section 3, it is possible to compare the accuracies and see that Logistic Regression with the OneVsRestClassifier performs better than the Random Forest classifier. However, the over- and undersampling methods do not seem to improve the accuracy of this classifier by much, while when testing with the Random Forest classifier, there was a general improvement of 0.04 in accuracy with the RandomUnderSampler method. This could be due to the parameter in Logistic Regression of "class_weight" which already takes into account the imbalanced nature of the data.

To further understand what the classifier is doing, it is important to analyse its results and this can be done with a number of visualisation techniques shown in subsections 4.1, 4.2 and 4.3 of this paper.

4.1 LIME Visualisations

LIME (Local Interpretable Model-Agnostics Explanation) is a machine learning package which creates visualisations which help us understand which features contribute to how the classifier classifies the data. Figure 2. shows an example of a misclassification where the classifier gave the highest probability to "Community Service (Volunteer)" whereas the true class was "Service Learning". However, it is interesting to note that the second highest probability was given to "Leadership Education", which holds some similarities to "Service Learning". "Service Learning" had the 4th highest probability. In the LIME representations, the category is shown and under it the features which contributed towards its probability; on the right are features which made it more likely to be that class and on the left are features which made it less likely, according to the classifier. The "Other" under Prediction Probabilities in the figures below, refers to being something other than the top 4 specified classes.

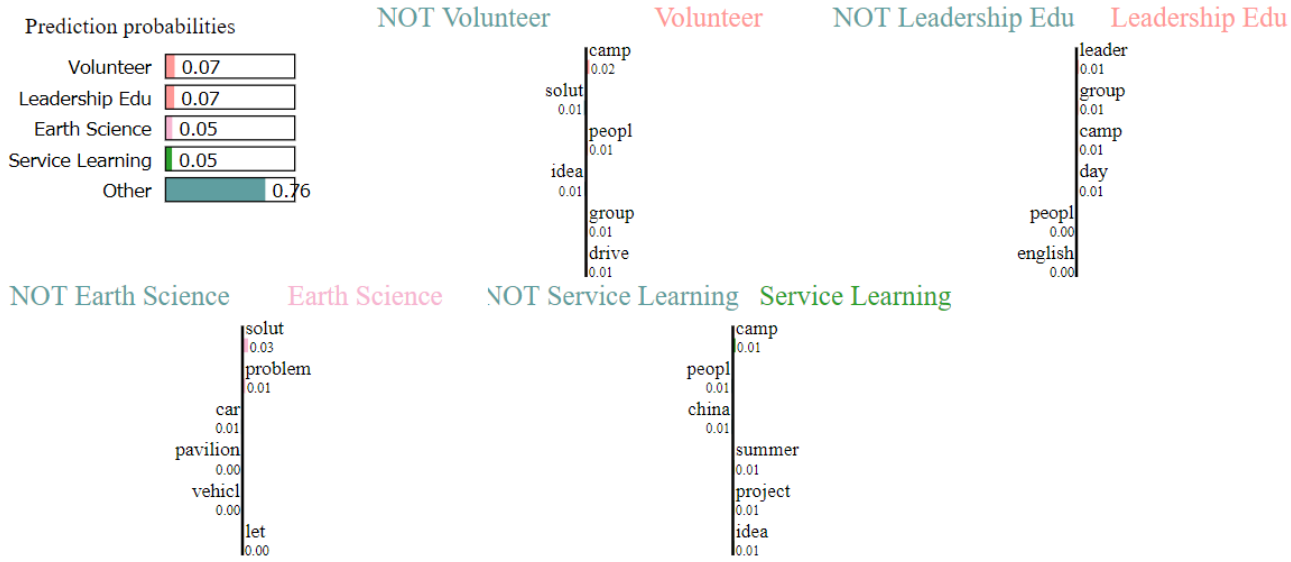


Figure 2.. LIME representation of the strongest features used to classify the first accomplishment in the test data. Example of misclassification.

Figure 3. is an example where the the classifier was able to correctly classify the accomplishment.

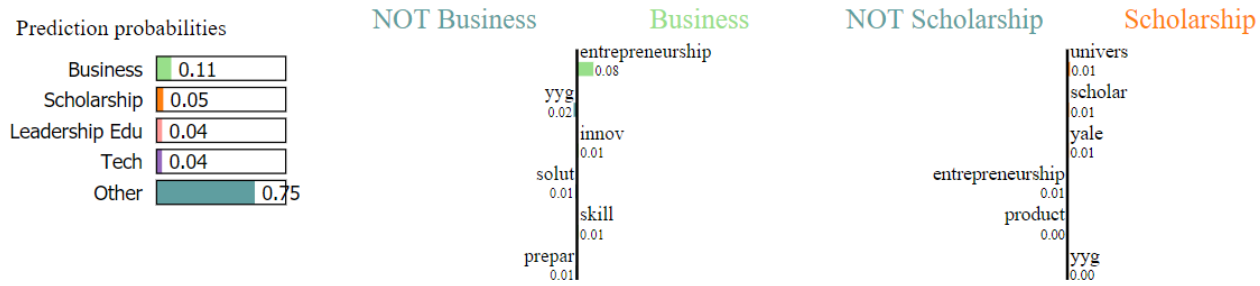
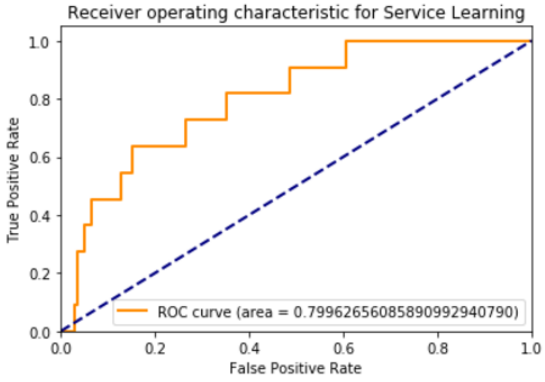


Figure 3.. LIME representation of the strongest features used to classify the first accomplishment in the test data. Example of correct classification.

4.2 ROC Curves

ROC (Receiver Operating Characteristic) curves are graphs which show the ratio of true positive rate (TPR) versus false positive rate (FPR) of classifications at different thresholds. The true positive rate is the ratio of correct classifications for the analysed class over all the accomplishments in that class, while false positive rate is the number of wrongly classified accomplishments as the class being analysed, over all the accomplishments which are not of that class. The area under the ROC curve can give us a numerical understanding of the ratio between true positive and false positive rate. Since the true positive rate is measured on the y-axis and the false positive rate on the x-axis, the larger the area of the curve the greater the true positive rate and the smaller the false positive rate. The graph can be used to pick the ideal tradeoff between TPR and FPR, by choosing a point on the curve, as each point refers to a specific classification threshold. The average area of all the classification's true and false positive rates is 0.96.

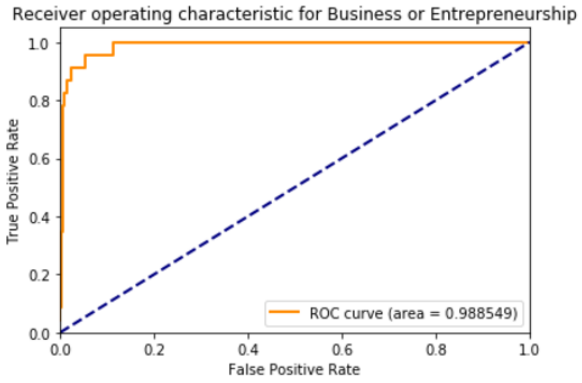
Graph 2. shows the ROC curve for "Service Learning", which has the smallest area.



Graph 2. ROC curve for "Service Learning". Area under the curve is 0.799.

From the confusion matrix, "Service Learning" is mostly wrongly classified as "Community Service (Volunteer)", while other classes are usually not mistaken for "Service Learning".

Graph 3. shows the ROC curve for "Business or Entrepreneurship Education". "Business or Entrepreneurship Education" has a very high true positive rate, since the classifier is able to correctly categorise accomplishments which fall into this class, however, it does have a few false positives.

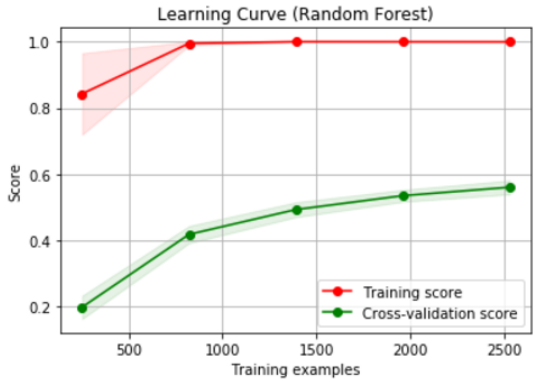


Graph 3. ROC curve for "Business or Entrepreneurship Education". Area under the curve is 0.989.

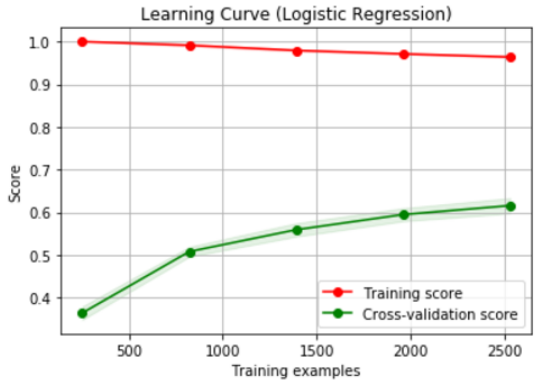
It is important to note that a few of the classes have an ROC curve with an area equal to 1.0, but this is usually due to there being only 1 to 3 examples of the class in the testing data. The classifier manages to correctly identify these classes, and not mistake other examples for these classes, but it is difficult to know how well the classifier can identify these minority classes without more examples.

4.3 Learning Curves

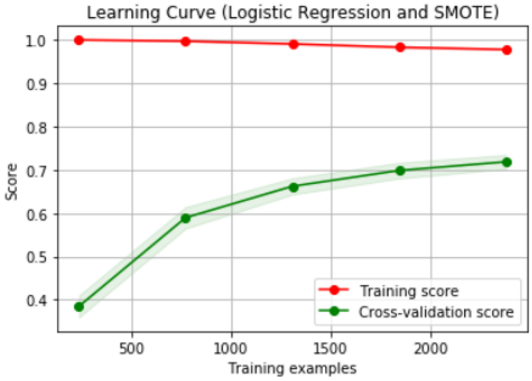
The last visualisation used to analyse the results were learning curves. These are graphs which analyse the accuracy score of the classifier depending on the amount of training data used. Learning curves can then make a prediction on how having more data will affect the accuracy of the classifier. In the graphs, the training score is the accuracy of the classifier on the training data and the Cross-validation score is the accuracy of the testing sample. The y-axis shows the accuracy score, while the x-axis are the amount of examples used in the training data.



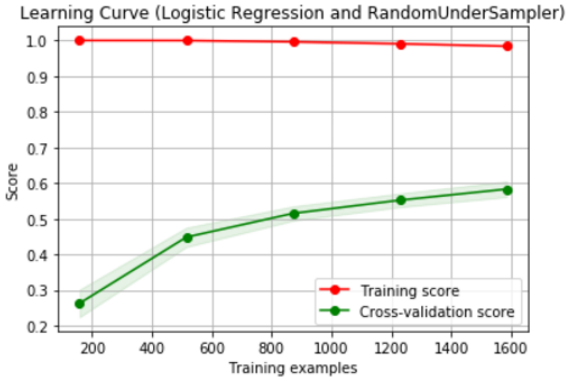
Graph 5. Learning curve of the Random Forest classifier.



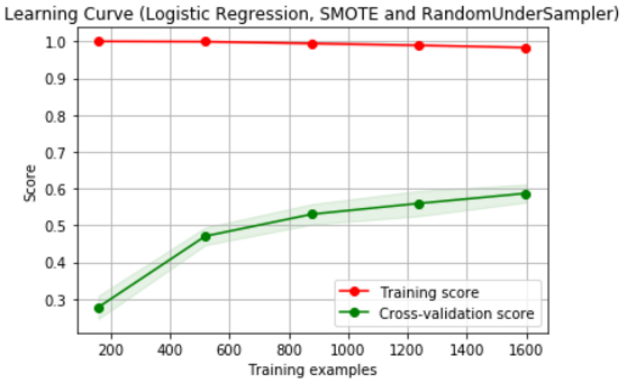
Graph 6. Learning curve of the Logistic Regression classifier.



Graph 7. Learning curve of the Logistic Regression classifier using SMOTE to oversample the training data.



Graph 8. Learning curve of the Logistic Regression classifier using RandomUnderSampler to undersample the training data.



Graph 9. Learning curve of the Logistic Regression classifier using SMOTE and RandomUnderSampler to over- and undersample the training data, respectively.

From the learning curves we use the extension of the Cross-validation score and Training score and where they would converge to predict how the classifier would improve with more data. From Graph 5. we can tell that using Random Forests, the training data seems to overfit and that the classifier might only improve to just lower than 0.6 accuracy. The most promising learning curve is from Graph 7. which uses the Logistic Regression classifier and SMOTE to oversample the training data. This shows that what might be most useful is not only more data, but rather more balanced data.

5 Conclusion and Suggestions

From the testing and analysing of the of the data and the results of the classifier, there are two main details which, if adjusted, would be able to improve the classifier.

Firstly, the imbalanced nature of the data. Although, "class_weight", SMOTE and RandomUnderSampling can mitigate the effect of the imbalanced data, it is not sufficient when there

are some categories which have fewer than 6 examples and therefore cannot be oversampled, and other categories with not enough examples for the classifier to find the right patterns for.

Secondly, some categories are too vague or too similar to other categories, and so use very similar language. For example, "Service Learning" and "Activism (Lobbying, Impactful Protesting, Policy)" are often categorised as "Community Service (Volunteer)", and "Debate/Speech" and "Leadership Education" as "Government or Politics". In these cases the classifier has a bias towards the majority classes in the training data. More balanced data would again help mitigate this, but also more consistent labeling from the application processors would help the classifier identify the right patterns. There are some accomplishments which could be more than one category, and this means that it is up to the grader to decide which category that accomplishment fits in better. This type of decision making differs from grader to grader. Sometimes accomplishments did have more than one label, and in this case the first grader's categorisation was kept.

With more balanced data, we would predict that the classifier could reach an accuracy around 0.7. To gain greater accuracy, the main suggestion would be to increase the threshold probability for classifying an accomplishments, that is, rather than the accomplishment being labeled as the category which it has the highest probability to be, according to the classifier, that the threshold be raised to over 0.5, 0.6, or even 0.8, so that the classifier only keeps the labels for the accomplishments it is most sure about. In this way, it will label the easily identifiable accomplishments, and leave the more ambiguous ones unlabeled, for application processors to categorise.

References

Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. **The NumPy Array: A Structure for Efficient Numerical Computation**, Computing in Science & Engineering, **13**, 22-30 (2011), DOI:10.1109/MCSE.2011.37

John D. Hunter.**Matplotlib: A 2D Graphics Environment**, Computing in Science & Engineering, **9**, 90-95 (2007), DOI:10.1109/MCSE.2007.55

Wes McKinney. **Data Structures for Statistical Computing in Python**, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. **Scikit-learn: Machine Learning in Python**, Journal of Machine Learning Research, **12**, 2825-2830 (2011)