COMP3009 Data Mining

Assignment Report

Kristian Rados (19764285)
Semester 2, 2021

Abstract

Throughout this assignment, I have taken the approach to try and automate as much of the process as possible so that I do not have to manually comb through the dataset to clean it and pick out the most relevant attributes. It is my belief that this strategy has resulted in satisfactory trained models. Even if that is not the case, however, it is clear that this approach, especially if improved slightly with some software engineering concepts, could be made into a very useful tool for solving this specific kind of data mining problem in the future, even if just for myself.

When training the models, I wanted to use dimensionality reduction with Principal Component Analysis to continue the process of removing attributes that weren't useful. However, I found that this in fact reduced the performance consistently, which surprised me. The one positive impact on accuracy which surprised me the most though was the hyperparameter grid search, which I assumed for whatever reason would not be particularly effective.

Data Preparation

Irrelevant Attributes

Irrelevant attributes are those which either: simply do not contribute enough identifying information relevant to the class label to be worth the added sparsity the extra dimension brings to the dataset, which may negatively influence model performance due to the curse of dimensionality (see: Wikipedia, 2021), or are redundant due to repeating information too similar to that already provided by other attributes.

There are three particular things which I checked for that resulted in an attributed being defined as irrelevant and consequently dropped:

1. Extremely low variance between numeric values.
2. Having only one unique value, whether categorical or numeric.
3. Being highly correlated with another attribute.

For point 1, the code would check for any numeric attributes that have a variance value below a predetermined threshold, which in this case I set to 0, meaning that only attributes with variance 0 would be removed. I set it to 0 as the variance values of the remaining numeric attributes were very similar, as can be seen in Figure 1, so I didn't know where to draw the line. Because all attributes with variance 0 were converted to categorical prior to this and removed in the next step, *this resulted in no attributes being removed.*

Similarly, the number of unique values in each categorical attributes, as seen in Figure 2, was checked. If there was only a single unique value, the attribute was deemed irrelevant and dropped. Just as with a very low variance, having one repeated value means that the attribute isn't bringing any valuable information to the classification. *This removed attributes 'C10', 'C15', 'C17', and 'C30'. Attribute 'ID' was also manually removed* as it had no useful information, simply representing the file's line number plus one.

C1	0.039618
C4	0.030299
C9	0.024297
C16	0.016566
C19	0.024764
C25	0.030437
C31	0.024297

Figure 1 (above): The variance values of all numeric attributes remaining at execution.

Class	2
C2	2
C3	5
C5	10
C6	5
C7	4
C8	4
C10	1
C12	4
C13	3
C14	4
C15	1
C17	1
C18	3
C20	4
C21	5
C22	2
C23	4
C24	3
C26	5
C27	4
C28	4
C29	2
C30	1

Figure 2 (right): The number of unique values per categorical class at execution.

Figure 3 (below): A correlation matrix for all numeric attributes at execution.

Finally, the correlations between numeric attributes were assessed, as can be seen in Figure 3. If two attributes are highly correlated, as indicated by white in the correlation matrix, it means they are likely representing the same information, so one is redundant. A threshold for removal of 90% correlation was set, which means it *removed attribute 'C25'*.

Duplicates

Duplicate attributes and instances are a problem as they

may mislead the final model into overemphasising the importance of particular features, resulting in overfitting or simply less accurate results in general.

Pandas and Numpy were used to identify, then drop, which instances were duplicates of prior ones. *This removed every instance between ID 900 and 999 (inclusive).*

C7	C12		C21	C26
V1	V1		V2	V2
V1	V1		V4	V4
V2	V2		V3	V3
V1	V1		V2	V2
V4	V4		V3	V3
V2	V2		V2	V2
V4	V4		V2	V2
V2	V2		V5	V5

Figure 4: An example of two sets of duplicate attributes in the original dataset.

The same method was used to identify and drop duplicate attributes, except the dataframe was first transposed so that each attribute was represented by a row and could be analysed automatically by Pandas. *This removed attributes 'C12', 'C26', and 'C31'.*

Missing Entries

Missing entries cause issues for the classifier models used and must be either removed from the dataset or imputed with new values (Sci-kit (Sci-kit Learn, 2021). The first step is to completely remove any attributes that mostly consist of missing values. To do this I picked a threshold of 80%, for which all attribute above that threshold were removed. *This removed attributes 'C11' and 'C32'.*

It wouldn't be suitable to remove attributes that only have a few missing attributes, as you could be removing a lot of information that would be helpful for the classifiers. Hence, for attributes with fewer than 5% missing values, the data was imputed with the attribute's mean value, for numeric, or mode value. *This imputed values for attributes 'C3', 'C4', 'C13', and 'C29'.*

There were no attributes between 5 and 80% missing values, but if that were the case a different scheme may have been used for that case.

	C11	C32
0	NaN	NaN
1	NaN	NaN
2	NaN	NaN
3	NaN	NaN
4	NaN	NaN
...	...	...
1095	NaN	NaN
1096	NaN	NaN
1097	NaN	NaN
1098	NaN	NaN
1099	NaN	NaN

Figure 5: Mostly missing attributes.

Data Types

It is important that each attribute in the dataset is assigned the correct datatype, either numeric or categorical. This is because there may be many attributes which were actually categorical, but because those categories were represented in the dataset with numeric values they may be incorrectly interpreted as being ordinal attributes by the classifiers, misleading the models as to how those values should be treated. Figure 6 shows an example of such attributes.

To start with, the *'Class' attribute was converted to categorical*, as the class of an instance in a classification problem is, by definition, a categorical value. As for the rest of the dataset's initially numeric attributes, the number of unique values that appears in each was used to determine if it should be considered numeric or

categorical. A threshold of 12 was chosen, so that any attribute with 12 or fewer distinct values was converted. Such a low number of values makes for a safe assumption and also makes sure to include any potential attributes representing months of the year or ages discretized into 10-year bins. *This converted from numeric to categorical attributes 'C15', 'C17', 'C20', 'C23', 'C27' and 'C29'.*

	C15	C17	C20	C23	C27	C29
0	0.0	1.0	4	1	4	1.0
1	0.0	1.0	2	2	2	2.0
2	0.0	1.0	3	1	4	1.0
3	0.0	1.0	4	1	2	1.0
4	0.0	1.0	4	1	4	1.0
...	...	...	...	...	...	...
1095	0.0	1.0	2	1	3	1.0
1096	0.0	1.0	4	1	3	1.0
1097	0.0	1.0	4	2	4	1.0
1098	0.0	1.0	3	1	3	1.0
1099	0.0	1.0	2	1	2	1.0

Figure 6: Attributes with few distinct values.

Finally, to make sure that categorical attributes aren't assumed to always be ordinal, one-hot encoding is used to convert each distinct category of each categorical attribute into its own binary true-false attribute. This unfortunately increases the sparsity of the dataset significantly by adding many new attributes. However, it removes the ordinality of the attributes, which is important as there is no way of knowing whether any of the attributes can be treated as ordinal without domain knowledge and the classifiers must accept the dataset in the form of a numeric matrix.

Data Transformation

Either scaling or standardisation needs to be applied to the numeric attributes of the dataset, as each has their own scale and distribution, which could make it difficult for the classifier model to accurately learn representations. A large difference in attribute scale, as can be seen in Figure 7, may cause the model to overemphasise the importance of the attribute with the larger values.

Scaling was used rather than standardisation to ensure that all values are within the range [0,1], so that the differences in scale shown do not negatively influence model accuracy. The downside of this as opposed to standardisation is that outliers may have an outsized impact on this new scale. However, I believe the effect of this is not dire in this case and it would be preferable to avoid negative values.

I also attempted to reduce the dimensionality of the model even further by using Principal Component Analysis. However, I found that this reduced performance on all models each time I attempted to use it. I am unsure as to whether this was because of my incompetence or because all the remaining attributes were too important to remove. It requires further investigation.

	C1	C9	C19
0	53	7865	3824
1	35	3904	5160
2	40	4296	3720
3	28	1402	6245
4	40	1503	5496
...	...	...	...
1095	31	6758	4200
1096	39	2674	4844
1097	23	2123	4044
1098	25	589	5054
1099	34	2576	5034

Figure 7: Attributes with high values prior to scaling/standardisation may be unfairly prioritised.

Data Classification

Class Imbalance

A problem with the provided dataset is that there was an imbalance in the set of labelled. As can be seen in Figure 8, class '0' is represented far more than class '1'. This would likely result in the classifier being able to classify class '0' well but not class '1', dragging down overall model performance.

This issue is by ameliorated by using the SMOTE algorithm to oversample the underrepresented class, which is class '1' in this case.

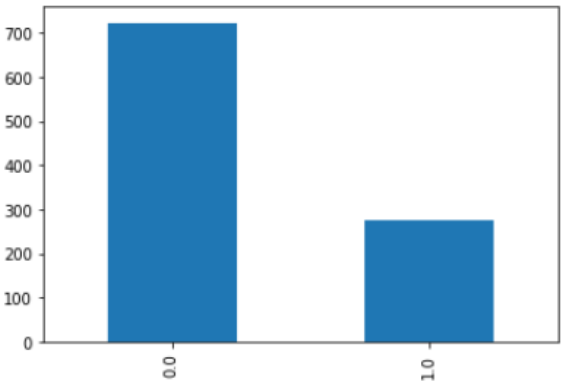

Figure 8: Class imbalance in labelled dataset.

Put simply, this algorithm will generate new instances with attribute values that result in them geometrically being placed along the lines joining the instances of the original dataset. This ensures that the generated samples remain within the distribution of the original data. SMOTE used to oversample the minority class until it was equal in size to the majority class through the model's cross-validation pipeline, which means that it was applied for every single fold of cross-validation rather than prior to training as a data preparation step. This was to prevent any leakages of information from the training folds to the validation folds.

Model Training and Tuning

To continue the automation theme, I made sure to automate as much of the hyperparameter tuning process as was reasonable. This was done by using a grid search to train a model on every combination of hyperparameters as specified by the few remaining user-adjustable hyperparameters that weren't part of the search. For any algorithms with a random component a fixed seed of 7 was used to ensure reproducible results.

K-nearest Neighbours Classifier

The parameters shown in Figure 9 were searched. `kmin` and `kmax` represented the user-adjustable parameters, which I set assigned the values 1 and 21 respectively. A max of 21 was chosen as I never saw a value higher than 15 used by the search's best models.

```
k_range = list(range(kmin, kmax + 1, 2))
params = {
    'model__n_neighbors' : k_range,
    'model__weights'     : ['uniform', 'distance'],
    'model__metric'      : ['manhattan', 'euclidean', 'chebyshev']
}
```

Figure 9: Parameters used for the K-nearest Neighbours Classifier grid search.

The best model found by the grid search an `n_neighbors` value of 5, the 'distance' `weights` scheme, and, rather surprisingly to me, the 'chebyshev' `metric`.

It achieved a *mean validation accuracy of 64.9%.*

Decision Tree Classifier

The parameters shown in Figure 10 were searched. `min_samples_min` and `min_samples_max` represented the user-adjustable parameters, which I set assigned the values 1 and 15 respectively. A max of 15 was chosen as I never saw a value higher than 12 used by the search's best models.

```
min_samples_leaf_range = list(range(min_samples_min, min_samples_max + 1))
params = {
    'model__criterion'        : ['gini', 'entropy'],
    'model__splitter'         : ['best', 'random'],
    'model__min_samples_leaf' : min_samples_leaf_range
}
```

Figure 10: Parameters used for the Decision Tree Classifier grid search.

The best model found used a `min_samples_leaf` value of 12, the 'entropy' `criterion`, and the 'random' `splitter`.

It achieved a *mean validation accuracy of 69.7%*.

Naive Bayes Classifier

Only the parameter shown in Figure 11 was searched, as I found no other suitable parameters to tune for the probabilistic model. The max value for `var_smoothing` is 1 as a value greater than that would not make sense given that the data only has values in the range [0,1].

```
params = {
    'model__var_smoothing' : [1e-10, 1e-9, 1e-8, 1e-7, 1e-6, 1e-5,
                              1e-4, 1e-3, 1e-2, 1e-1, 1.0]
}
```

Figure 11: Parameter used for the Naive Bayes Classifier grid search.

The best model found used a `var_smoothing` value of 1e-1.

It achieved a *mean validation accuracy of 62.8%.*

Multi-layer Perceptron Classifier

I chose to use a Multi-layer Perceptron classifier, more generally known as a neural network, mostly because I personally find them and the deep learning revolution that they have spawned very interesting. However, because the training dataset is was so small I didn't have high expectations. It seems that those expectations were incorrect.

The parameters shown in Figure 12 were searched. I did not make use of user-adjustable parameters in the same way I did with the K-nearest Neighbours and Decision Tree classifiers. Instead, I engaged in a trial and error process for determining which parameters and what respective values to include in the search. This is because there are far too many hyperparameters to consider, with my initial parameter set resulting in a search that would train over 90,000 models. Here is a summary of the changes I made during my process of trial and error:

- I initially included `'model__learning_rate' : ['constant', 'invscaling', 'adaptive']` and `'model__learning_rate_init' : [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]`. After a few searches I realised that the default values were used just about every time, so they were removed.
- `'sigmoid'` was removed from `'model__activation'` as it had similar performance to `'tanh'`, albeit it appeared to be slightly worse.

- `'model__alpha'`, the regularisation parameter, initially included 10 different values, each differing by an order of magnitude from the rest. However, I soon realised that only 1e-4 and 1e-3 ever appeared in the best models.
- A wide array of different layer architectures were included at various points as part of the `model__hidden_layer_sizes` parameter, including (10), (50), (50,10), (100,50), (150), etc. For whatever reason, only models with size 100 layers appeared in my best models, so the rest were removed.

```
params = {
    'model__hidden_layer_sizes' : [(100), (100,100), (100,100,100)],
    'model__activation'         : ['tanh', 'relu'],
    'model__solver'             : ['lbfgs', 'sgd', 'adam'],
    'model__alpha'              : [1e-4, 1e-3]
}
```

Figure 12: Parameters used for the Multi-layer Perceptron Classifier grid search. Searching this reduced parameter set took 260 seconds to train utilising all 8 cores of my mid-range desktop computer's CPU.

It achieved a *mean validation accuracy of 73.5%.*

Cross-Validation Scheme

I decided to use a cross-validation scheme as opposed to a fixed validation dataset as the size of the labelled dataset isn't particularly large — a bad arbitrary choice of validation dataset could have a non-trivial effect on performance. I chose to use 9 folds for cross-validation so that there could be 100 instances per fold, just like the test set. This is because the labelled dataset after data preparation only has 900 instances remaining. SMOTE is used as part of the model pipeline, so each validation fold will have 50 instances of each class, once again like the test set.

Model Comparison

I have decided to compare the models using both accuracy and F1-score. An F1-score represents the harmonic mean of the model's precision and recall, which both represent the model's performance in regards to false positives and false negatives respectively. As compared to accuracy, the F1-score takes the distribution of the data into account, more accurately portraying the performance of the model for imbalanced test data.

The two models will be picked according to a custom comparison score: a weighted average of the accuracy and F1-score according to the formula $\frac{2*accuracy+1*f1}{3}$. I am emphasising accuracy over F1-score because the test set is balanced and, due to our lack of domain knowledge, there are no indications of any particular need to avoid false negatives more than usual. It is important to note that this is just a comparison metric which is meaningless on its own.

Comparison Scores

K-Nearest Neighbour Classifier: 0.555

Decision Tree Classifier: 0.617

Naive Bayes Classifier: 0.597

Multi-layer Perceptron Classifier: 0.665

Due to their higher scores, the Multi-layer Perceptron Classifier and Decision Tree Classifier were thus chosen as my two classifiers for the final prediction.

Prediction

Finally, the predictions are made on the unlabelled test set. To do this, the pipelines for the best two models selected above, which includes their hyperparameters and the SMOTE oversampling step, are taken and used to train two new models on the entirety of the labelled training dataset with no validation. This is done to ensure the model's used for final predictions have use all the information available to use to make the most accurate prediction.

Two sets of predictions for the test set are then made, one for each model, with those results then being written to the `results.csv` file for future reference and for submission here. Those results can be seen in Figure 13.

ID	Predict1	Predict2	ID	Predict1	Predict2	ID	Predict1	Predict2	ID	Predict1	Predict2
1001	0	0	1026	0	0	1051	0	0	1076	1	1
1002	1	1	1027	1	1	1052	0	0	1077	0	0
1003	0	0	1028	0	1	1053	1	0	1078	0	0
1004	0	0	1029	0	0	1054	1	0	1079	0	0
1005	0	1	1030	0	0	1055	1	0	1080	1	1
1006	0	0	1031	0	0	1056	0	0	1081	0	0
1007	1	0	1032	1	0	1057	1	1	1082	0	0
1008	0	0	1033	1	1	1058	1	1	1083	0	1
1009	0	1	1034	0	1	1059	1	1	1084	1	0
1010	0	1	1035	1	1	1060	0	0	1085	1	0
1011	0	0	1036	0	1	1061	0	1	1086	0	0
1012	0	0	1037	1	0	1062	1	1	1087	0	1
1013	1	1	1038	1	0	1063	0	0	1088	0	0
1014	1	0	1039	1	1	1064	1	0	1089	0	0
1015	0	0	1040	0	0	1065	0	1	1090	0	0
1016	0	0	1041	1	0	1066	1	0	1091	0	0
1017	0	0	1042	1	0	1067	1	1	1092	1	1
1018	1	1	1043	0	0	1068	0	0	1093	1	1
1019	0	0	1044	0	1	1069	0	0	1094	0	0
1020	1	0	1045	0	0	1070	0	0	1095	0	0
1021	0	1	1046	0	1	1071	0	0	1096	1	1
1022	1	1	1047	0	0	1072	0	0	1097	0	0
1023	0	0	1048	0	0	1073	0	0	1098	1	1
1024	0	0	1049	1	0	1074	0	0	1099	0	0
1025	0	0	1050	0	0	1075	1	0	1100	0	0

Figure 13: The final test set predictions made by the two chosen models.

Estimated Accuracy

I estimate the combined test set prediction accuracy of my models to be 69.6%.

This value was calculated by taking the average validation set accuracy of my two chosen models and subtracting 2%. The reason for this subtraction is to account for any possible overfitting and to generally be conservative given that we had no domain knowledge available to help with data preparation. I have not subtracted from my validation accuracy to account for any imperfect oversampling, as I am unsure as to how accurately SMOTE will represent the true data distribution for the training set's underrepresented class.

$$\frac{73.5 + 69.7}{2} - 2.0 = 69.6\% \text{ predicted test set accuracy}$$

Conclusion

This assignment has taught me a lot about data science practices and traditional machine learning, providing me a very satisfying baseline as somebody looking to explore the more advanced deep learning field further. I have found the validation results that I have obtained to sufficiently satisfactory to my own personal standards. I purposely wrote my data preparation and training script to be extensible and usable for future data mining projects, with it having many opportunities for abstraction, extension, and usability improvements. Because of this, I am confident that the work done and skills learned here will serve me well for any future projects in this field and those adjacent to it.

References

Sci-kit Learn developers. 2021. "Imputation of missing values." Sci-kit Learn. https://scikit-learn.org/stable/modules/impute.html

Wikipedia. 2021. Curse of dimensionality: Machine Learning. Wikimedia Foundation. https://en.wikipedia.org/wiki/Curse_of_dimensionality#Machine_Learning