CPSC 4660 Project Report: Crime Rate Prediction Using an Enhanced Naive Bayes Algorithm

David Neufeld, Sahil Devnani, Cameron Nickle, Vishnu Undhad

CPSC 4660

Implementation Summary

In this project, we performed preprocessing on a dataset, implemented two algorithms for the purpose of creating prediction models, and ran those models so that we could perform evaluations on the performance of the algorithms. The dataset that we chose contained crime statistics for Vancouver from 2003-2017. All of our implementation was written in Python, and we ran the implementation in a Jupyter Notebook. The first of the two algorithms that we implemented was Naive Bayes. The second algorithm that we implemented was a Recursive Feature Elimination (RFE) algorithm that used a Random Forests algorithm as the feature selector. The results from the RFE algorithm were utilized to modify the data before it was fit using Naive Bayes. The purpose of implementing and running these algorithms on the dataset was to see how effective the models that resulted from our algorithms would be at predicting a crime type based on time and location data.

Implementation Details

The first stage in our implementation required obtaining our dataset. The dataset that we chose (Crime in Vancouver | Kaggle) was found on Kaggle, and we loaded it directly into our Kaggle environment to work on it. Once the data was in the environment, we imported it into our Python code using a Pandas DataFrame. At this point, we were able to perform

preprocessing on the data so that it could become useful and compatible with the algorithms we used to generate our models.

The first step in our preprocessing involved removing any rows where the crime type attribute was ‘Offence Against a Person’. This was done because location and specific time data was missing in these rows. Next, we removed any rows that contained N/A (or None, etc.) values in order to ensure that our dataset was complete. After ensuring our dataset was complete, we found the four highest frequency crime types and narrowed our dataset down to only include rows with those values. With the dataset narrowed down, we decided to derive two new attributes: ‘DayOfWeek’ and ‘WeekOfYear’ (we did this because it was also done in the paper our project was based on, and they stated that it may be beneficial for the algorithms). Next, we replaced the crime types with numerical values so that the data could be compatible with the algorithm. We also removed the ‘HUNDRED_BLOCK’ and ‘NEIGHBOURHOOD’ attributes since they contained non-numerical data and converting the data to numerical data had not proven useful in a trial run. Finally, the dataset was split into four parts: a training dataset with the target, a testing dataset with the target, a training dataset with the remaining attributes, and a testing dataset with the remaining attributes. After completing our preprocessing, we implemented our two algorithms.

The first algorithm that we implemented was Naive Bayes, a classification algorithm that takes in a subset of data with the target(s) and a subset of data with the remaining attributes. The interface of this algorithm was based off of the interface described for GaussianNB in scikit-learn (sklearn.naive_bayes.GaussianNB — scikit-learn 1.2.2 documentation). However, our implementation is more basic and we included only functions that were needed for our purposes. The two main functions for this algorithm are NBfit and NBpredict.

The NBfit function takes in the target and remaining attribute subsets of a dataset to compute mean, variance, and priors. The setup for this function involves obtaining the number of rows and columns from the remaining attribute subset of data and obtaining the unique classes and number of classes from the target subset. Matrices of the size of the number of classes by the number columns obtained earlier are created and filled with zeros. Similarly, an array of the size of the number of classes filled with zeros is created. These matrices are then filled with useful data in a for loop that runs as many times as there are classes. The first matrix stores the mean values for each attribute in the remaining attributes subset, the second matrix stores the variance values for each attribute, and the array stores the priors (the number of rows for a given class divided by the total number of rows in the dataset). These values are all kept in the class.

The NBpredict function takes in a dataset with all the attributes other than the target and utilizes the values calculated in the fit function to perform its predictions. Two other class functions are used within this function, _NBpredict and NPGaussian. The NBPredict function creates an array of predictions by calling _NBpredict for every row in the dataset provided and storing the return values of that function in the array. _NBpredict takes in a row of a dataset and calculates the probability that the row would be classified as each of the classes; this is done in a loop for each class. The summation of the logs of the array returned by NPGaussian called on the row with the class is added to the log of the prior for this class (this is the probability). Once the probability has been calculated for each class, the class with the highest probability is returned as the classifier for that row. The NPGaussian function that is called takes in the index of the class (for the array holding the classes) and the row. It then calculates a gaussian equation using the mean and variance of the class and returns it.

The second algorithm that we implemented was Recursive Feature Elimination (RFE). RFE recursively calls an estimator algorithm capable of calculating feature

importances and removes the lowest feature(s) (attribute(s)) until the desired number of features has been reached. As was the case with our implementation of Naive Bayes, our implementation of RFE is based off of the interface for RFE found in the scikit-learn library (sklearn.feature_selection.RFE — scikit-learn 1.2.2 documentation). Again, our implementation is more basic and only includes the functions that were needed for our purposes. The algorithm consists of the following functions: an initializer, a fit function, a transform function, and a fit_transform function (the fit_transform function simply calls fit and transform sequentially on the same parameters and was not used in our evaluation).

The RFE object is initialized with optional parameters for the number of features to select and the step size (the number of features to remove per call of the estimator algorithm). Within the initializer, the estimator algorithm is set by default to be the RandomForestClassifer algorithm from scikit-learn since we were specifically interested in the results of RFE on Naive Bayes when the Random Forest algorithm is used. The initializer sets the number of features to select to be either the number specified or sets it to 0 if no value is specified (the fit algorithm will handle this further when it is called). It also initializes the step size to the size specified or 1 if no value is provided. Several other attributes are initialized to default values for later use.

The fit function takes in two subsets of the dataset (again, the target and the non-target attributes). If the number of features was set to 0 or less, then it sets the number of features to select to be half of the features in the dataset. It then sets the ranking and support arrays (both of which contain information regarding which features are selected) to be the same size as the number of features in the non-target dataset. The function then calls RandomForestClassifer with the subsets to determine the feature importance values and it removes the number of features that were defined by the step size. This process is repeated, but the non-target dataset that is given to the estimator has the lowest feature(s) removed from it. This process repeats

until the desired number of features has been reached. During this loop, the ranking array is continually updated with the ranks (starting at lowest rank first). At the end of the loop, any unranked features are set to 1 (meaning they are the chosen features). This array is then used to update the support array which has true values for each chosen feature and false for all other features.

The transform function takes in a dataset with non-target attributes. It then checks each feature with the support array. If the support array has a true value for the index of that feature, then it is kept, otherwise, the feature is dropped from the dataset. When the function has checked all features, the new dataset with the features dropped is returned.

Evaluation Strategy

The evaluation of the two algorithms took into consideration the same metrics that were described in our proposal. The first metric that we checked was accuracy: how often was the model produced by our algorithm correct. The second metric that we checked was precision: the fraction of our predictions that were accurate. The third metric we checked was recall: the number of correct predictions of instances of crime over the total instances of crime. Lastly we checked the F1 score: the mean of the second and third metric stated above.

Beyond the metrics described above, we also decided to perform a comparison between the models that resulted from our algorithms and the models that resulted from the sci-kit learn algorithms. The table containing the metrics described in the last paragraph contains six columns and four rows. The columns are: scikit-learn Naive Bayes, our Naive Bayes, scikit-learn Naive Bayes with scikit-learn RFE, our Naive Bayes with scikit-learn RFE, scikit-learn Naive Bayes with our RFE, and our Naive Bayes with our RFE. The rows are simply the metrics described in the last paragraph.

Evaluation Results

The table below compares all six models that were used for crime prediction. They are each compared on accuracy, precision, recall, and F1 score. We will now discuss the results seen in this table.

	SK NB	Our NB	SK NB w/ SK RFE	Our NB w/ SK RFE	SK NB w/ Our RFE	Our NB w/ Our RFE
Accuracy	0.493628	0.436023	0.465498	0.480840	0.479631	0.425618
Precision	0.412902	0.404622	0.181253	0.120210	0.347540	0.331506
Recall	0.356008	0.390670	0.256250	0.250000	0.318783	0.332515
F1_Score	0.341525	0.331569	0.191399	0.162354	0.288146	0.290744

We will begin by discussing the highest scores for each metric. For three out of the four metrics, the scikit-learn Naive Bayes model scored highest out of all six. These metrics are accuracy, precision, and F1 Score with a 49.36, 41.29 and 34.15. However, our Naive Bayes model scored highest on one metric: recall with a score of 39.07. Our Naive Bayes algorithm lagged just behind the scikit-learn Naive Bayes on precision and recall, but had a score almost 6 lower for accuracy. We note here that all the highest scores were held by original Naive Bayes models.

Since all the highest scores were held by Naive Bayes models, that means that RFE did not appear to aid in producing better results overall. However, narrowing in on the case of our Naive Bayes algorithm combined with the sci-kit learn RFE algorithm reveals that, while all other metrics were lower, it increased the accuracy of our Naive Bayes by almost 4.5. It can also be noted that our RFE algorithm did not cause precision, recall, and F1 score to drop nearly as drastically as the scikit-learn version did and that the combination of scikit-learn Naive Bayes with our RFE algorithm produced a higher level of accuracy than scikit-learn Naive Bayes with scikit-learn RFE.

Now, we will narrow in on the results that came strictly from our own implementations. On every metric, our Naive Bayes algorithm performed better without RFE than with. Accuracy dropped by just over 1, precision by over 7, recall by nearly 6, and F1

score by over 4. These results are not what was expected. The RFE was expected to improve model accuracy by eliminating features that could skew results. These results are the opposite of what was found in the paper our project was based on.

The results of our evaluation were disappointing, but they were not without reason. Naive Bayes, and Naive Bayes with RFE, can produce good results when provided with the appropriate data. Our dataset may have had issues, and the preprocessing of the data may have been done incorrectly. This would be able to explain the overall poor scores for each of the metrics. Since the original algorithm did not score well, it is not a surprise that RFE did not aid in correcting that. However, from our own evaluation, we have to conclude that RFE does not improve the results of the Naive Bayes model, but rather, it has a negative effect on the results.

Conclusions Drawn From the Experience

Overall, this project was enjoyable and we were excited to get some hands-on experience with a new programming language that contains more powerful libraries than C++. Performing data pre-processing proved challenging at times, but it felt worth it when the model was finally able to take in the dataset. Implementing the algorithms also caused some issues, but again, when they were finally running it seemed to pay off.

While the project was enjoyable, if we could start over, there is one thing we would do differently: we would choose a much smaller dataset. Working with a dataset of over half a million rows meant that it was impossible to know all the data contained in the dataset. This made pre-processing a bigger hassle than it would have needed to be. Also, using such a large dataset with slightly more intensive models, such as RFE, meant that the time it took to run a model was much longer than it could have been.