Nikhil Narang
German Credit Risk Analysis
EE 559: Mathematical Pattern Recognition (DEN)
Email: nnarang@usc.edu
2 May 2017

Abstract

The objective of this project was to develop a pattern recognition system to minimize the risk experienced by German banks when determining the creditworthiness of loan applicants. The dataset includes 1000 data samples made up of a mixed feature set of 20 categorical and numerical features. In order to develop an optimal pattern recognition system, I started with an exploratory data analysis to visually identify data patterns and areas for data simplification. I applied preprocessing by encoding categorical data into numerical values, accounting for missing data, and developing a method for standardizing data. For the purpose of comparison, I designed a baseline classifier using only priors. I then developed three more advanced classifiers: (1) Gaussian Naïve Bayes, (2) k-Nearest Neighbor, and (3) Support Vector Machine. After a series of rigorous 10-fold cross validation on the training data (80%), the Support Vector Classifier was selected and applied to the remaining test data (20%). Ultimately, the Support Vector Classifier achieved an accuracy of 80.0% on the test data with an F1 score of 0.86 for class 1 and 0.64 for class 2.

Introduction

The objective of this project was to develop a pattern recognition system that would minimize the risk and maximize the profit of a bank by providing a decision rule to approve a candidate for a loan. The dataset used in the pattern recognition system provides socioeconomic and demographic features for 1000 previous loan applicants, and is classified into good credit risks (S_1) and bad credit risks (S_2). Thus, the project involves solving a 2-class decision problem with a mixed feature set of categorical and numerical features.

This implementation uses the Python scikit-learn library along with pandas and numpy. Initial testing was conducted on the Kaggle dataset, but the results observed in this report were observed by applying classifiers to the full German Credit Risk dataset from the UCI Machine Learning Repository. My approach in this project involved the following series of steps:

(1) Exploratory Data Analysis
(2) Feature Preprocessing
(3) Feature Selection and Dimensionality Reduction
(4) Baseline Classification
(5) Improved Classification with Naïve Bayes, SVM, and KNN Classifiers
(6) Performance Evaluation with Cross Validation
(7) Interpretation of Results

A section has been devoted to each of these steps in order to provide a detailed understanding of the thought process behind the development of the classification system.

Exploratory Data Analysis

It was extremely important to understand the nature of the features in the dataset. This section provides an analysis of the full German Credit Risk dataset from the UCI Machine Learning Repository. The first two tables explore the frequency of attributes of categorical data types.

ATT 1: Status of Checking Account	**A11: Below 0 DM**	**A12: Below 200 DM**	**A13: Above 200 DM**	**A14: No Checking Account**	
	273	269	63	394	
ATT 3 Credit History	**A34: Critical Account/ Other Credits Exist**	**A33: Delay in Paying**	**A32: All Credits Paid Until Now**	**A31: All Credits Paid Back Duly**	**A30: All Credits Paid Back or None Taken**
	292	88	530	49	40
ATT 6 Savings Account/Bonds	**A65: No Savings Account**	**A61: Below 100 DM**	**A62: Between 100 DM and 500 DM**	**A63: Between 500 DM and 1000 DM**	**A64: Above 1000 DM**
	182	603	103	63	48
ATT 7 Length of Current Employment	**A71: Unemployed**	**A72: Less than 1 year**	**A73: Between 1 and 4 years**	**A74: Between 4 and 7 years**	**A75: More than 7 years**
	62	172	339	174	252
ATT 8 Installment Rate	**Above 35%**	**25% to 30%**	**20% to 25%**	**Below 20%**	
	136	231	157	476	

ATT 9 Personal Status and Sex	A91: Male Divorced	A93: Male Single	A94: Male Married	A92: Female
	50	547	92	310
ATT 10 Other Debtors or Guarantors	A101: None	A102: Co-Applicant	A103: Guarantor	
	906	41	52	
ATT 12 Assets	A124: No Property	A123: Car	A122: Life Insurance	A121: Real Estate
	154	332	232	281
ATT 14 Concurrent Credits	A141: Bank	A142: Stores	A143: None	
	139	47	813	
ATT 15 Type of Housing	A151: Rent	A152: Own	A153: Free	
	179	712	108	
ATT 16 Number of Credits at Bank	1	2 or 3	4 or 5	More than 7
	633	333	28	6
ATT 17 Occupation	A171: Unskilled/Unemployed	A172: Unskilled Resident	A173: Skilled	A174: Highly Skilled
	22	200	629	148
ATT 19 Telephone	A191: Yes	A192: No		
	596	403		
ATT 20 Foreign Worker	A201: Yes	A202: No		
	962	37		

ATT 4 Purpose	A40: Car (New)	A41: Car (Old)	A42: Furniture/ Equipment	A43: Radio/ Television	A44: Domestic Appliances	A45: Repairs	A46: Education	A48: Retraining	A49: Business	A410: Other
	234	103	181	279	12	22	50	9	97	12

The tables above provide a visualization of the categorical data from the provided dataset. In some cases, originally numerical categories like Number of Credits at Bank or Installment Rate have been converted into categorical data because they are better represented as categories. Additionally, the categories can be subdivided into three types: (1) binary categorical, (2) unordered categorical, and (3) ordered categorical. These three types are treated differently during the preprocessing portion of the pattern recognition system. By visualizing the categorical data in the tables above, certain attributes were merged, thus applying some level of "smoothing" to the dataset. Merged categories are indicated in light green. This smoothing process was applied to attributes that were well correlated together as indications of good or bad credit, or if an attribute had too few samples to provide significant information relating to the overall classification.

The table below provides a visualization of the numerical features in the German Credit Risk dataset. We clearly see some level of asymmetry in the numerical features; in fact, most of the numerical features appear to be positively skewed. Nevertheless, we will assume a Normal distribution will provide an effective approximation of the probability distributions of the numerical data.

Statistic	ATT 2 Duration of Credit	ATT 5 Credit Amount	ATT 11 Duration in Current Residence	ATT 13 Age	ATT 18 Number of Dependents
Count	1000	1000	1000	1000	1000
Mean	20.903	3271.258	2.845	35.546	1.155
SD	12.058814	2822.736876	1.103718	11.375469	0.362086
Min	4	250	1	19	1
25%	12	1365.5	2	27	1
50%	18	2319.5	3	33	1
75%	24	3972.25	4	42	1
Max	72	18424	4	75	2

The numerical features are assigned a numerical type, and if no categorical ranges are specified, they are treated as continuous features during classification. The feature types and attribute merging of categorical features are configured in the *metadata.json* file. A Metadata class, defined in *defines.py*, imports and parses the JSON file into metadata attributes that are used throughout the pattern recognition system.

Feature Preprocessing

Feature Encoding

The previous section explored some of the initial restructuring of the original dataset, but after these values were imported into the pattern recognition system, they had to be preprocessed into types that could be easily accepted by the classification models. This feature preprocessing depended largely on the type of the feature:

- Binary Categorical: features encoded into $[0, 1]$
- Ordered Categorical: features encoded into $[0, n]$, where n is the number of categories
- Unordered Categorical: features encoded into n-width vectors using a one-hot encoding, where n is the number of categories
- Numerical (with range categories): features encoded into $[0, n]$, where n is the number of range categories
- Numerical (without range categories): no encoding applied

The actual encoding of categorical data is specified in the *metadata.json* file, but is actually applied to the data in the *encode(...)* function defined in *preprocessing.py*.

Handling Missing Values

After all the data had been re-cast into numerical values, it became necessary to deal with missing values. Missing values were dealt with differently in the German Credit Risk dataset provided by Kaggle and the dataset provided directly from the UCI Machine Learning Repository.

Since the Kaggle dataset was initially "cleaned up," some elements were missing and needed to be estimated before classification methodologies could be applied. While this report displays results from the full UCI dataset, I implemented each of these strategies during my initial testing and compared the results. The following four strategies were tested:

1. *Delete samples with missing data.* This strategy was a good place to start, but ultimately 1000 samples is already quite small, so the removal of these additional samples reduced the dataset too significantly to make this a worthwhile strategy for handling missing data.
2. *Replace missing data with feature statistics*. This strategy involved computing the mean, median or mode from the training set and then applying this value to all missing elements in the training and test sets. The median statistic appeared to provide the best estimation for missing values
3. *Replace missing data using a probability distribution*. This strategy involves developing a probability distribution for features with missing values and then using random sampling to assign values to the missing samples.
4. *Replace missing data using k-Nearest Neighbors*. This strategy involved applying k-Nearest Neighbors based on the values of all the other features for a sample with missing data. This strategy was applied with $k = 10$, and appeared to provide a high level of accuracy in replacing missing data. Ultimately, the issue with this approach is its application on test data

In the UCI dataset, missing values were handled using a different approach. The "missing" data as indicated in the Kaggle dataset were indicated with certain Unknown or Not Applicable attributes in the full UCI dataset, so on the onset, there were no missing values. The Kaggle dataset handled these Unknown or Not Applicable attributes by removing them, but my approach was to preserve these attributes and encode and eventually classify samples with these attributes intact. For Ordered Categorical features like "A14" or "A65," I analyzed the distribution relative to the classification of the samples to decide how they should be ordered relative to the other attributes. In the example of "A14: No Checking Account," I observed that samples with this attribute were generally more likely to be of good credit, which seems to indicate such a candidate might be likely to have a checking account at another bank, and thus a higher order should be assigned. On the other hand "A65: Unknown/No Savings Account" samples were more likely to be of bad credit, which indicates a lower order should be assigned. This approach to handling missing values provided a high level of accuracy, since in most cases these Unknown attributes actually provided some level of information in how a sample might be classified.

Feature Standardization

Feature standardization is important in order to assure the proper behavior of certain machine learning algorithms. Given non-standardized data, some algorithms may fail to provide accurate or precise results. In this pattern recognition system, feature standardization is applied by removing the mean and scaling to unit variance. Centering and scaling are applied independently to each feature by computing statistics from the training set and then transforming both the training and testing sets.

Certain cases exist where the application of a unit variance is undesirable. This is particularly the case for classifiers that conduct dimensionality analysis using Principal Component Analysis, which relies on the largest variances to choose directions. In these cases, the mean is removed from the dataset.

Feature standardization in this system was particularly complex because of the heterogeneous nature of the dataset. In order to implement correct standardization, binary and categorical values should be skipped during the standardization process. A custom *NumericScalar* transformer is defined in *scaler.py* in order to perform standardization on numerical features without available range categories.

Feature Selection and Dimensionality Reduction

Recursive Feature Elimination (RFE)

Feature Selection is an extremely important part of classification systems for a number of reasons. First, feature selection results in a model that is easier to interpret due to the smaller feature size. Second, feature selection has the ability to reduce overfitting of a decision boundary by reducing the variance in the model. And third, feature selection reduces the complexity of a system, thus reducing the operating time. My pattern recognition system utilizes Recursive Feature Elimination for Feature Selection.

RFE is a feature selection algorithm that treats feature selection like a search problem. The goal of RFE is to select features recursively by considering a smaller and smaller set of features. This is achieved by ranking each feature according to some external estimator and then removing the k lowest ranking features. This recursive process is continued until the reduced feature set reaches a desirable number.

In this system, the built-in RFE class from scikit-learn, *sklearn.feature_selection.RFE*, is applied to the KNN and SVM classifiers in order to reduce the feature set. Both implementations use a Random Forest Classifier estimator, from *sklearn.ensemble.RandomForestClassifier*. The Random Forest Classifier is a tree-based strategy that naturally ranks features by how well they improve the "purity" of a node. Features that improve purity the most are ranked higher on the tree and features that improve it the least are ranked lower, so pruning trees below a given node allows for efficient feature selection.

Principal Component Analysis (PCA)

Dimensionality Reduction is another way of reducing the number of features while preserving the overall nature of the original dataset. PCA is a statistical procedure that uses orthogonal transformations to convert a set of observations of possibly correlated variables into a set of linearly uncorrelated variables. This is done by finding the eigenvectors of a covariance matrix with the highest eigenvalues and then projecting the data into a new subspace of equal or fewer dimensions. Thus, PCA reduces the number of features by constructing a smaller set of features that captures a significant portion of the information found in the original feature space.

In this system, the built-in PCA class from scikit-learn, *sklearn.decomposition.PCA*, is applied to the Gaussian Naïve Bayes classifier in order to reduce the feature set.

Baseline Classification

A baseline classifier was developed as a standard of comparison with other classification techniques. The baseline classifier is based solely on the known priors of the system:

$$P(S_1) = \frac{n_1}{N} = \frac{700}{1000} = 0.7 \quad and \quad P(S_2) = \frac{n_2}{N} = \frac{300}{1000} = 0.3$$

We can represent this baseline classifier with a Bernoulli distribution with an outcome $k = 1$ (success) with a probability $p = 0.7$ and an outcome $k = 0$ (failure) with probability $q = 1 - p = 0.3$. Therefore, the probability density function of the baseline classifier is given by:

$$f(k; p) = \begin{cases} p & if\ k = 1, \\ 1 - p & if\ k = 0. \end{cases}$$

This classifier is equivalently implemented in Python by generating a random number of range $[0,1)$ and returning a 1 if the number is less than 0.7 and a 2 if the number is greater than or equal to 0.7. Thus, baseline classifier constructs a set of predicted classifications based on the initial priors, and the values are compared with actual classifications to generate accuracy and F1 scores.

A custom *BinomialClassifier* estimator is defined in *binomial.py*. The baseline classifier is implemented in the *baseline_classifier(…)* function in *model.py*. This function implements 10-fold cross validation on the baseline classifier in order to generate scores with the same scrutiny as the rest of the classifiers.

Improved Classification Techniques

Gaussian Naïve Bayes

The Gaussian Naïve Bayes Classifier uses a Naïve Bayes decision rule while assuming the class conditional density is a Normal Gaussian distribution. Thus, this classifier applies the decision rule:

$$p(\underline{x}|S_1)P(S_1) \underset{S_2}{\overset{S_1}{\gtrless}} p(\underline{x}|S_2)P(S_2)$$

where the class conditional density is assumed to be normal:

$$p(\underline{x}|S_k) = \frac{1}{(2\pi)^{D/2}\left|\underline{\underline{\Sigma}}_K\right|^{1/2}} \exp\left\{-\frac{1}{2}(\underline{x}-\underline{m})^T \underline{\underline{\Sigma}}_K (\underline{x}-\underline{m})\right\}$$

and the priors are assumed by the relative frequencies of each class in the training set.

In this system, the scikit-learn Gausian Naïve Bayes class, *sklearn.naive_bayes.GaussianNB*, is used to implement the classifier. The classifier is implemented in the *naïve_bayes_classifier(…)* function in *model.py*. This classifier is implemented as part of a pipeline that standardizes the feature set (without applying unit variance), applies PCA for dimensionality reduction, and then applies the Gausian Naïve Bayes classifier to the resulting feature space. The only hyperparameter in this classifier is the feature dimensions that result from PCA. The hyperparameter is selected using a cross-validation based grid search over the entire space of possible dimensions.

k-Nearest Neighbor (kNN)

k-Nearest Neighbor classification uses a nearest neighbor density estimation technique to assign class membership. In kNN, a sample is assigned to the class most common among its k nearest neighbors. If $k = 1$, a sample is assigned to the class of its single nearest neighbor in feature space. This is essentially a discriminative model for classification, which uses a decision rule based posterior probabilities:

$$\hat{P}(S_1|\underline{x}) \underset{S_2}{\overset{S_1}{\gtrless}} \hat{P}(S_2|\underline{x})$$

where the posterior probabilities can be estimated by:

$$\hat{P}_n(S_i|\underline{x}) = k_n^{(i)}/k_n$$

This results in a simplified decision rule based on the frequency of samples of various class memberships:

$$k_n^{(1)} \underset{S_2}{\overset{S_1}{\gtrless}} k_n^{(2)}$$

In this system, the scikit-learn k-Nearest Neighbors class, *sklearn.neighbors.KNeighborsClassifier* is used to implement the classifier. The classifier is implemented in the *knn_classifier(…)* function in *model.py*. The classifier is implemented as part of a pipeline that standardizes the feature set, applies RFE for feature selection, and then applies the kNN classifier to the resulting feature space. The kNN classifier has a single hyperparameter given by the number of neighbors, k. This hyperparameter is selected using a cross-validation based grid search over the entire feature space of possible dimensions.

Support Vector Machine (SVM)

Support Vector Machines are supervised learning classifiers that perform non-linear mapping to expanded feature space, add equality and inequality constraints to a problem, and can operate on sparse data. An SVM model maps data in order to separate data of different classes by a clear gap that is as wide as possible. SVM is highly efficient for non-linear classification using the kernel trick, which involves performing a non-linear mapping of inputs into high-dimensional feature space.

In this system, the scikit-learn Support Vector Class, *sklearn.svm.SVC*, is used to implement the classifier. The classifier is implemented in the *svm_classifier(…)* function in *model.py*. This classifier is implemented as apart of a pipeline that standardizes the feature set, applies RFE for feature selection, and then applies the SVM classifier to the resulting feature space. The SVM classifier is fixed with an RBF kernel, but hyperparameters exist for the kernel coefficient, gamma, and the penalty parameter, C. The hyperparameters are selected using a cross-validation based grid search over a fixed space of gamma and C parameters.

Performance Evaluation

Cross Validation and Dataset Usage

Evaluating estimator performance is one of the most important parts of building a classification system. If a system was to learn a decision rule based on a set of training data, and then test on the same data, the classifier would observe a perfect accuracy despite the fact that it may fail on unseen data. This is a problem of overfitting, and it can be solved by using cross-validation.

In this system, I initially separate 20% of the data as a test set and the remaining 80% as the training data. The training data used to train and test all four classifiers from the previous section: (1) baseline classifier, (2) Gaussian Naïve Bayes classifier, (3) kNN classifier, and (4) Support Vector classifier. These classifiers are tested using a cross-validation loop in which the training data is randomly separated into k-folds and then trained on $k - 1$ folds while the remaining fold is used for testing. This is repeated k times such that each fold is used for testing at least once. The average accuracy and F1 scores from the cross-validation loop are stored and the best performing classifier (based on accuracy) is returned. The best performing classifier from each of the four types are compared, and the one with the highest accuracy is applied to the test set.

In this system, cross validation for model selection is performed using the built in scikit-learn class given by *sklearn.model_selection.GridSearchCV*. This class exhaustively searches over a grid of specified parameters, thus performing cross validation and returning the best performing hyperparameters.

Classification Performance

The cross validation approach results in two sets of results. The first set of results is the average accuracy and F1 scores associated with each of the classifiers during cross validation. In this system, I opted for an exhaustive 10-fold cross validation scheme which resulted in the following accuracies and F1 scores:

Training Set Parameters	Baseline Binomial Classifier	Gaussian Naïve Bayes Classifier	k-Nearest Neighbors Classifier	Support Vector Machine
Accuracy	0.555	0.757	0.762	0.772
F1 Score (S_1)	0.67	0.84	0.86	0.90
F1 Score (S_2)	0.33	0.48	0.50	0.66

The system chooses the best performing classifier based on during training. The k-Nearest Neighbors classifier

Test Set Parameters	Support Vector Machine
Accuracy	0.800
F1 Score (S_1)	0.86
F1 Score (S_2)	0.64

The selected hyperparameter values for each of the classifiers are displayed in the tables below:

Hyperparameter Selections	Gaussian Naïve Bayes Classifier
pca__n_components	8

Hyperparameter Selections	Support Vector Machine
knn__n_neighbors	15

Hyperparameter Selections	k-Nearest Neighbors Classifier
svm__C	1
svm__gamma	0.1

Interpretation of Results

The classifiers developed in this project provide a relatively strong indication of whether a candidate is credit-worthy or not, with a final accuracy of 80.0% with the Support Vector Machine classifier. The following table shows the relative improvement of each of the classifiers as observed during training when compared to the baseline. In all three cases, a significant improvement has been made on the baseline model, so the classifiers must provide some improvement in credit prediction.

Training Set Parameters	Baseline Binomial Classifier	Gaussian Naïve Bayes Classifier	k-Nearest Neighbors Classifier	Support Vector Machine
Improvement	1.000	1.364	1.373	1.391

In order to actually visualize what this improvement in credit prediction might look like, it may be helpful to evaluate these results in terms of actual profit and loss experienced by the bank. This can be done by making a few assumptions:

- Assume a True Positive prediction results in 27% profit over 3 years
- Assume a False Positive prediction results in 100% loss over 3 years
- Assume False Negative and False Positive result in 0% profit over 3 years

From these assumptions, we can design a cost matrix that will help us analyze the classification results in terms of total loss and/or profit obtained by the bank:

Actual	Predicted	
	0.800	
	S_1	S_2
S_1	0.27	0
S_2	-1.00	0

The True Positive and False Positive results from the confusion matrices that resulted from each of the classifier cross validation sequences are displayed in the table below. If we apply the following equation to these results, we can obtain a unit profit value per DM loaned. If we multiply this value by an average loan amount of 5000 DM and a total number of loans of 1000, we obtain the total profit values in DM. These results are displayed in the following table:

$$profit = (cost_{FP})(\%_{FP}) + (cost_{TP})(\%_{TP})$$

Actual	Predicted			
	Baseline	GNB	KNN	SVM
	S_1	S_1	S_1	S_1
S_1	0.448	0.65	0.679	0.698
S_2	0.178	0.176	0.18	0.139
Unit Profit (per DM)	-0.05704	-0.0005	0.00333	0.04946
Total Profit (DM)	-285200	-2500	16650	247300

These results clearly indicate that that the more complex KNN and SVM classifiers would allow the bank to make a profit under the assumptions.

Appendix: German Credit Risk Classifier Code Review

Code Overview

All the necessary code has been placed in the *src* repository. The following list provides an overview of the main modules in the repository and their function:

- binomial.py – defines a *BinomialClassifier* class for the baseline classifier
- defines.py – defines constants and Metadata class
- enum.py – defines an enum function
- exploration.py – initial data exploration sequence
- main.py – runs all classifiers, selects the best, and runs the best classifier on test set
- metadata.json – UCI Repsoitory German Credit Risk dataset contents description
- model.py – defines all four classifiers
- preprocessing.py – provides util functions for loading and encoding data
- scaler.py – defines a custom *NumericScaler* function

Running the code requires the following pre-installed libraries:

- scikit-learn
- pandas
- numpy

All the code that was written for this project is original. I did use the scikit-learn and Pandas documentation (cited) for the purpose of learning about the libraries, but none of the code in this project is directly copied from an external source.

Trace of Classification Sequence

Once the libraries are installed, the following sequence can be used to run a full classification sequence:

```
>> cd src
>> python main.py
```

The resulting trace is displayed below. For the purpose of repeatability, all random value seeds have been set to constants, so the above sequence should print this same trace.

```
Loading data file from location: '../data/Proj_dataset_2.csv'
Parse complete: 20 features and 1000 samples

Encoding categorical data into numerical values
Column 'Checking Account Balance': categorical data mapped to feature values [0, 1, 2, 3]
Column 'Credit History': categorical data mapped to feature values [0, 1, 2]
Column 'Purpose': unordered categorical data mapped to 7 new features
Column 'Savings Account Balance': categorical data mapped to feature values [0, 1, 2, 3, 4]
Column 'Length of Current Employment': categorical data mapped to feature values [0, 1, 2, 3, 4]
Column 'Installment Rate': numerical data mapped to feature values [0, 1, 2, 3]
Column 'Personal Status': unordered categorical data mapped to 2 new features
Column 'Applicant Type': categorical data mapped to feature values [0, 1]
Column 'Property': unordered categorical data mapped to 4 new features
Column 'Concurrent Credits': unordered categorical data mapped to 3 new features
Column 'Housing': unordered categorical data mapped to 3 new features
Column 'Number of Bank Credits': numerical data mapped to feature values [0, 1]
Column 'Occupation': categorical data mapped to feature values [0, 1, 2, 3]
Column 'Telephone': categorical data mapped to feature values [0, 1]
Column 'Foreign Worker': categorical data mapped to feature values [0, 1]
```

```
Encoding complete: 34 features and 1000 samples

Applying baseline classification
accuracy: 0.555
f1_score: 0.668

             precision    recall  f1-score   support

          1       0.71      0.63      0.67       571
          2       0.29      0.38      0.33       229

avg / total       0.59      0.56      0.57       800

Applying Naive Bayes classification
Fitting 10 folds for each of 34 candidates, totalling 340 fits
[Parallel(n_jobs=1)]: Done 340 out of 340 | elapsed:    3.3s finished
accuracy: 0.757
Best parameters set:
       pca__n_components: 8

             precision    recall  f1-score   support

          1       0.79      0.91      0.84       571
          2       0.63      0.38      0.48       229

avg / total       0.74      0.76      0.74       800

Applying k-nearest neighbor classification
Fitting 10 folds for each of 50 candidates, totalling 500 fits
[Parallel(n_jobs=1)]: Done 500 out of 500 | elapsed:  5.3min finished
accuracy: 0.762
Best parameters set:
       knn__n_neighbors: 15

             precision    recall  f1-score   support

          1       0.79      0.95      0.86       571
          2       0.75      0.37      0.50       229

avg / total       0.78      0.79      0.76       800

Applying SVM classification with RBF kernel
Fitting 10 folds for each of 25 candidates, totalling 250 fits
[Parallel(n_jobs=1)]: Done 250 out of 250 | elapsed:  2.7min finished
accuracy: 0.772
Best parameters set:
       svm__C: 1
       svm__gamma: 0.1

             precision    recall  f1-score   support

          1       0.83      0.98      0.90       571
          2       0.90      0.52      0.66       229

avg / total       0.85      0.84      0.83       800

Best classifier: svm
       accuracy: 0.800

             precision    recall  f1-score   support

          1       0.78      0.96      0.86       129
          2       0.88      0.51      0.64        71

avg / total       0.81      0.80      0.78       200
```

Works Cited

Albon, Chris. "Notes on Data Science, Machine Learning & Artificial Intelligence." *Data Science Machine Learning and Artificial Intelligence*. N.p., n.d. Web. 02 May 2017.

"Analysis of German Credit Data." *Analysis of German Credit Data*. N.p., n.d. Web. 02 May 2017.

Hnyk, Daniel. "Creating Your Own Estimator in Scikit-learn." *Daniel Hnyk*. Daniel Hnyk, 21 Nov. 2015. Web. 02 May 2017.

"Pandas: Powerful Python Data Analysis Toolkit." *Pandas: Powerful Python Data Analysis Toolkit — Pandas 0.19.2 Documentation*. N.p., n.d. Web. 02 May 2017.

"Scikit-learn." *Scikit-learn: Machine Learning in Python — Scikit-learn 0.18.1 Documentation*. N.p., n.d. Web. 02 May 2017.

"Scikit-Learn Cheat Sheet." *Python for Data Science* (2012): 317-18. Web.