Machine Learning – Report 1

David Mendes (44934) & Francisco Cunha (45412)

October 21, 2017

1 Introduction

For this project we were given a data set from the UCI machine learning repository, which we proceeded to fit and test with three distinct classifiers: Logistic Regression, k-Nearest Neighbors and Naïve Bayes. This involved (but was not restricted to) parameter optimization, model fitting and validation. To finalize, we estimated each classifier's true error, and compared the classifiers using McNemar's test with a 95% confidence interval.

2 Classifiers and Parameter Optimization

a. Logistic Regression

Logistic regression is a classification algorithm that uses a logistic function to estimate probabilities and map each element of our input to some class of our (discrete) set of output classes.

This logistic function ("S"- shaped sigmoid curve) takes in an element and outputs a value between zero and one. In order to map this output to a discrete class, a threshold value (default of 0.5 in sklearn.linear_model#LogisticRegression) is used. This threshold will define the class to which the input given belongs to.

For this classifier, the parameter we optimized was C, a parameter used to control the level of regularization. With this technique, we attempt to solve overfitting by adding a penalty on the different parameters of the model. Depending on the value of C, those penalties will be more or less "strict".

There are various penalty norms available, but we ended up using sklearn.linear_model#LogisticRegression's default one, 'L2', resulting in a ridge regression.

It's important to notice that C, as stated in the official sklearn documentation, is the inverse of regularization strength. This means that the regularization growth (strictness/penalty weight) is inversely proportional to the growth of C.

So, how exactly does regularization affect the classifier?

- Higher regularization (lower value of C):
    - Increased penalty.
    - Reduced freedom and more resistance to outliers.
    - Smoother logistic curves.
    - Too much regularization can result in too strict fits - underfitting.

- Lower regularization (higher value of C):
    - Smaller penalties imply more "relaxed" fitting (and high sensibility to outliers).
    - Possibly very sharply sloped logistic curves.
    - None/too little regularization may cause overfitting.

As seen above, too much regularization may cause underfitting, and too none/too little regularization may cause overfitting. Thus, finding a good value for C implied finding the right balance between flexibility and restrictiveness.

b. K-Nearest Neighbors

K-Nearest Neighbors (or k-NN) is a lazy-learning algorithm. It's also an example of a non-parametric algorithm - even though it can have parameters (and it does, as we'll see below), it is not completely determined by the parameters. Yet, it does not make any assumptions on the underlying data distribution. This is rather useful, as for many practical applications most of the data doesn't necessarily follow the typical theoretical assumptions made (e.g. normally distributed, linearly separable ...).

How does k-NN work? Essentially, by keeping the labelled training set, the k-NN classifier will label a new point with the label of the majority of the k points in the training set that are closer to the new point.

There are various parameters that can be adjusted, the most relevant here being:

- k, the number of neighbors.
- metric, the distance metric to use.
- p, the power of the distance metric (in case the Minkowski metric is used).

Because we're dealing with continuous numerical features, we used the (default) Minkowski metric, or p-norm. We also used the default p value of 2, so that it would match the Euclidean distance.

Thus, the main parameter we optimized for this classifier was k. This should be an odd number when dealing with binary classification problems, to avoid ties. To understand exactly how the value of this parameter affects the classifier we must first understand the base case: k = 1.

For a 1-NN classifier (k = 1), new points are labelled with the label of the single closest point in the training set. By definition, using 1-NN (Voroni Tesselation) will mean a perfect fit of the training set - this is expected, as any point's nearest neighbor is itself.

However, this also means that our classifier will be extremely sensitive to outliers (non-representative points in the data set), which can translate into somewhat arbitrary classification boundaries, and thus high validation errors (overfitting).

As the number of neighbors used (k) increases, the classifier becomes less and less determined by local conditions, resulting in smoother decision boundaries. Following this logic, we can understand that over-extending this value will once again result in underfitting situations.

c. Naïve Bayes (with Kernel Density Estimation)

The Naïve Bayes classifier is the final supervised learning classification method that we implemented in this project. Essentially, it uses the probabilities of each feature belonging to each class to make a prediction. In contrast to the regular Bayes algorithm, the Naïve version assumes that the probability of each feature belonging to a given class is independent of all other attributes. This results in much simpler calculations, making it ideal to use in practice.

So, how do we train a Naïve Bayes classifier? We need to determine the conditional probability distribution of each feature given each class. Because we're working with features that have continuous values, we could've gone for either parametric or nonparametric models. Following the project's implementation guidelines, we went with a particular nonparametric model: kernel density estimation.

The model used would be the hearth of our "custom" Naïve Bayes Classifier, and thus it was crucial to choose and optimize its parameters well. We used the Kernel Density implementation from sklearn.neighbors, and focused on two closely-related parameters: kernel and bandwidth.

The kernel (function) is responsible for specifying the shape of the distribution placed at each point. To choose a kernel function, we analyzed the number of features and the number of observations. We had much less features than observations, and the size of the input data wasn't even that big. We thus ended up using the Gaussian kernel - while it boosted up the overall program's runtime considerably, it also gave us the smoother results.

However, it's important to notice that the quality of a kernel estimate depends less on the shape of the kernel than on the value of its bandwidth, which we shall call b. Whilst very narrow bandwidth values would lead to rather abrupt estimates (overfitting the training data), a bandwidth that's too big would lead to way-too-smooth estimations, most likely causing underfitting.

Likewise happened with the two previous classifiers, a good value for this parameter had to be one that balanced flexibility and correctness appropriately.

3 Cross-Validation

The optimal values for the three parameters mentioned above (C for Logistic Regression, k for K-NN and b for Naïve Bayes (with KDE)) were all found using the Cross-Validation technique. Cross-Validation is a great way to do parameter optimization (and thus, model selection), because it allows us to evaluate the actual models we're using, and not just specific hypothesis from each model.

More precisely, we used two thirds of the data to perform a stratified 5-Fold Cross-Validation. We resorted to sklearn.cross_validation's StratifiedKFold to rearrange (stratify) our data and ensure that each fold would be a good representative of the whole, as this generally leads to better results than regular cross-validation. The general procedure for each classification algorithm was similar:

For each outer iteration (where our parameter to optimize would vary) we calculated five folds. In a high level, each fold calculation involved:

- Instantiating a classifier with a specific parameter.
- Fitting the classifier with part of the training data.
- Calculating a training error.
- Calculating a validation error.
- Returning both the training and validation errors.

For each parameter iteration, we saved the average training and validation errors obtained (for the five folds). The training errors were simply used to plot how well each model adapted to the training data. The validation errors, however, were always compared with the previously lowest ones registered, in order to select the best model (determined by the value of the varying parameter, of course).

Naturally, we didn't use the validation errors to estimate the true error of our final classifier - if we did that, then this error would not be an unbiased estimate of the true error! That's the reason why we left out one third of the data for the test error measurements (and thus true error estimates). The training error (used to fit each model) and the validation error (used to select the best one) were both biased. Only the test error could give us an effectively unbiased estimate of the true error.

We used the fraction of incorrect classifications as the measure of the test error for each classifier. However - and even though the described high-level procedure was similar for every classifier - we used different strategies to calculate the training and validation errors for each classifier individually.

For the logistic regression we used the Brier Score. The Brier Score is a score function that measures the accuracy of probabilistic predictions. It is applicable to tasks in which predictions must assign probabilities to a set of mutually exclusive discrete outcomes. This effectively matches logistic regression for classification - logistic regression is still a regression model at hearth, and estimates probabilities using a logistic function, and so it makes sense to use the Brier Score here.

For the k-NN classifier, however, using the Brier Score wouldn't be appropriate at all, as it's not a regression-based classifier, nor it assigns probabilities. Thus, for this classifier we used the fraction of incorrect classifications as the measure of the error. We could have also used this strategy for our Logistic Regression classifier, but the Brier Score worked as well.

Finally, for the Naïve Bayes classifier we implemented our own custom measure of accuracy. This was rather simple, and essentially consisted of saving a classification's predictions and manually comparing them to the real output values. This way we could easily calculate the fraction of incorrect classifications.

4 Naïve Bayes Implementation

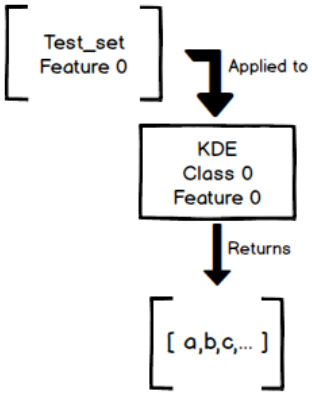
Given two data sets, one for training and one for testing / validation. We start by splitting the training set by classes and features. This way we end up with eight sets, result of the division of the original set (with four features and two classes). Each smaller set will be used to train a different Kernel Density Estimator.

We train each Kernel Density Estimator with a certain bandwidth; the bandwidth varies while doing cross-validation or represents the best bandwidth discovered in cross-validation when doing the final model evaluation. In a specific iteration all Kernel Density Estimators are trained using the same bandwidth value.

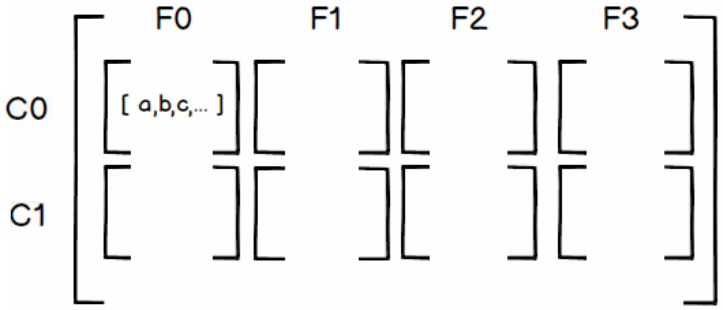
After training each Kernel Density Estimator we can obtain the relative likelihood of the value of a given feature matching a certain class.

To get the density logarithms from the estimators we need to split our data sets by features, each of these smaller sets will be applied to the kernels associated with the matching feature. Example: The set that contains the information relative to the first feature of the training set will be applied to the two Kernel Density Estimators that were trained for the first feature (with class 0 and class 1).

Because we want to obtain both the training error and the testing / validation error we need to apply to each Kernel Density Estimator two data sets relative to one single feature. The densities are calculated using the score_samples method.



Applying the Test_set of feature 0 (using the score_samples method) on the Kernel Density Estimator for class 0 and feature 0 will return an array with the calculated densities logarithms for each data point in the Test_set of feature 0. We can now store these values in the following manner.



Each smaller array contains the logarithms of the densities calculated using the previously demonstrated score_samples method.

We will have two of these structures, one for the training set and one for the testing / validation set.

We also determine the a priori logarithmic likelihood of a point belonging to each class on the training set, by counting the number of points that match each class and dividing it by the number of data points in the training set.

To classify a point, we can now determine the likelihood of it belonging to each class by determining the sum of the a priori class probability with the densities for each feature of that data point; the bigger value obtained will classify the point.

Meaning, if the sum of the a priori probability of class 0 with the densities of the point for each feature in class 0 is bigger than the sum of the a priori probability of class 1 with the densities of the point for each feature in class 1, then the point is classified as belonging to class 0, if not, class 1.

To determine the error associated with the model we compare the predicted and the true labels of the set we classified. The number of mismatches between the labels divided by the total amount of points will represent the error rate.

5 Testing and Classifiers comparison

We ran multiple tests and will present the results we found to be more representative.

The following results were recorded by running each classifier model ten times and averaging the results. For each parameter (c, k, b) we also present the value achieved in every iteration.

```
Average Results

C Values:  [512, 512, 128, 128, 64, 128, 128, 128, 128, 256]
Average C: 211
LR Average Accuracy: 98.79%

K Values:  [1, 3, 5, 5, 1, 3, 3, 7, 3, 3]
Average K: 3
KNN Average Accuracy: 99.91%

B Values:  [0.11, 0.11, 0.17, 0.09, 0.09, 0.15, 0.15, 0.09, 0.09, 0.25]
Best B: 0.13
NB Average Accuracy: 92.45%

McNemar's test between LR and KNN:  [4.0, 5.0, 9.0, 4.0, 0.0, 0.0, 2.0, 1.0, 2.0, 1.0]
Average McNemar's score between LR and KNN: 2.80

McNemar's test between LR and NB:  [11.0, 24.0, 13.0, 24.0, 19.0, 24.0, 10.0, 23.0, 29.0, 32.0]
Average McNemar's score between LR and NB: 20.90

McNemar's test between KNN and NB:  [25.0, 39.0, 32.0, 39.0, 26.0, 25.0, 20.0, 31.0, 40.0, 37.0]
Average McNemar's score between KNN and NB: 31.40
```

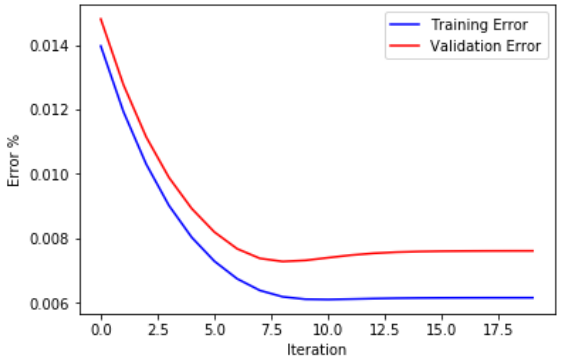
Considering the McNemar's Test we compared each classifier with a 95% confidence interval which brings the value on which we can reject the null hypothesis (that the two classifiers perform identically) to 3.84.

Given that fact we can clearly say that the Logistic Regression Classifier and the K Nearest Neighbors Classifier pulled ahead of the Naïve Bayes Classifier.

When comparing the Logistic Regression Classifier and the K Nearest Neighbors Classifier we, usually and on average get values lower than 3.84, meaning we can assume with 95% confidence that the classifiers perform identically.

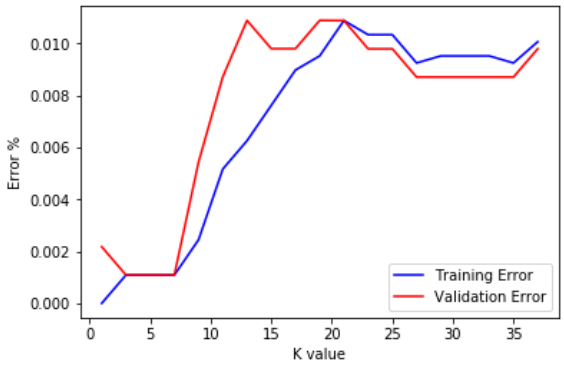
6 Error Plots

Logistic Regression Classifier



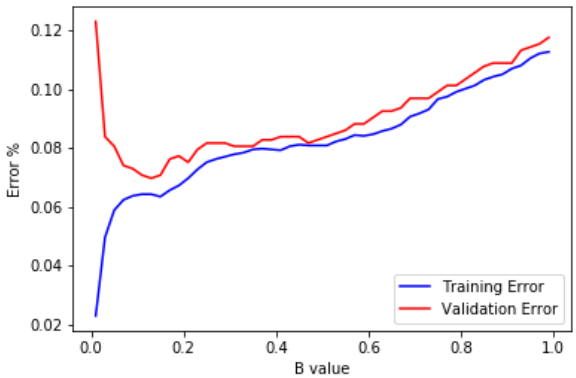

```
Best C: 256
Accuracy: 99.34%
```

K-Nearest Neighbors Classifier




```
Best K: 3
Accuracy: 99.78%
```

Naïve Bayes Classifier




```
Best B: 0.13
Accuracy: 93.16%
```

7 Conclusion and results analysis

After concluding our experiments, we achieved both results that we were expecting and results that were somewhat unexpected at first glance.

For the Logistic Regression Classifier, the results were easily explainable as the parameter floated around the middle range values proposed. Whilst lower values of C would result in high regularization and thus cause underfitting. On the other end of the spectrum high values of C would cause low regularization and lean towards overfitting.

In the K-Nearest Neighbors Classifier we had some struggle on understanding why the values of the parameter (k) always tended to be small in such an ample range. As the number of neighbors used (k) increases, the classifier becomes less and less determined by local conditions, resulting in smoother decision boundaries. Following this logic, we thought that over-extending this value could result in underfitting situations, what ended up occurring.

On the other hand, very small values of k, mean that the classifier will be very sensitive to outliers. This would usually translate into bumps that would create somewhat arbitrary classification boundaries, and thus high validation errors (overfitting). However, this wasn't the case. After various measurements, our values of k ranged between {1,3} with just some measurements achieving values slightly higher. The explanation we have for this phenomenon is that our data set has very, very few outliers, and is overall very consistent - hence why no overfitting scenarios were detected when using small values for k.

Regarding the Naïve Bayes Classifier, the results were in-line with the analysis performed in the previous models. We got small bandwidth values that in theory would lead to data overfitting. This fact corroborated our theory that our data set has a small number of outliers.

The results in the Naïve Bayes Classifier where also not surprising on the stand point of accuracy. We knew that this classifier made a very strong assumption on the fact that the features are conditionally independent given the class. This fact is not always true and can introduce prediction errors.

8 Bibliography

- https://en.wikipedia.org/wiki/Brier_score
- https://en.wikipedia.org/wiki/Logistic_regression
- https://en.wikipedia.org/wiki/Kernel_density_estimation
- http://scikit-learn.org/
- https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/
- Lecture Notes (http://aa.ssdi.di.fct.unl.pt/files/LectureNotes.pdf)