Najwa Laabid

Computing Assignment 2:
Report

Task 1:

- **Task description**

In this task, we were asked to implement two variants of the SVM algorithm: a linear and an RBF-kernel model. The two models were to be trained on the same dataset, with similar proportions (creditDE.csv, training proportion: .75). We would then use ROC-AUC to evaluate the two models and compare their performance.

Part 1 discusses the implementation and training of the SVM algorithms. Part 2 gives an overview of ROC-AUC curve, and describes the plotting procedure for linear SVM (with hard and soft margins) and RBF-kernel SVM (with sigma = 1.0). The last section (part 3) interprets the 3 ROC-AUC curves and what they tell us about the data.

- **Part 1: Linear and Kernel SVMs**

Linear SVMs, as the name suggests, are best suited for linearly-separable data. In the case of binary classification, SVMs seek to define a hyperplane separating the two classes while maximizing the margin between data points closest to the area of separation from either side, also known as *support vectors*. Linear SVMs come in two variants: hard-margin and soft-margin. The first type aims to keep 'clean margins' and works well for perfectly linearly-separable datasets. In practice however, such datasets are seldom encountered. As a result, the second variant of SVMs loosens the constraints on the SVM margins by allowing some data points to be close to the hyperplane, and therefore showing more tolerance for noisy datasets while giving reasonable separation ability.

When the data is not linearly separable, both variants would give poor results/fail to define the separating hyperplane. A work-around in this situation is to project the data points into a higher dimension, in which they become linearly separable, run the SVM algorithm as usual, then project the separating hyperplane back to the original dimension to get a different separating shape. This kind of projection tends to be costly computation wise, so we use kernel functions to estimate the transformations without performing them explicitly. Multiple kernel functions exist, notably: radial-basis or gaussian function (rbf), polynomial function, and hyperbolic-tangent (tanh) function.

The file *resources.py* provides implementations for linear SVMs and kernel SVMs with 2 kernel functions: polynomial and rbf. The modifications I made from assignment 1 were: changing the code of `compute_mulitpliers()` function (*line 509*) to handle the case of soft-margin SVM (with c != 0), and change the calculation of the bias in `prepare_svm_model()` (*line 535*) to use the average of support vectors instead of an arbitrary support vector. For preparing the training and testing datasets, I implemented the function `split_dataset()` (*resources.py*, line 186) which uses the code from Task 2 from Computing Assignment 1 to split the data. The training and testing of the models used the functions `prepare_svm_model()` and `svm_predict_vs()` as defined in *resources.py*.

- **Part 2: ROC-AUC**

Receiver Operating Characteristic (ROC) curve is a classification model evaluation metric. It allows the visualization of the trade-off between TPR (True Positive Rate, which can be interpreted as 'benefit') and FPR (False Positive Rate, which is a measure of the 'cost') [1]. Different TPR and FPR values are generated using various thresholds for the classification model. In the case of SVM, these thresholds are generated from the SVM scores for the test data points, by taking the lowest score (corresponding to the negative data point furthest away from the separating hyperplane) as the first threshold. This classifies all data points as belonging to the negative class, if we consider the classification condition to be (score >= t) => positive class. Subsequent threshold values are computed from the average of two consecutive data point scores. For every threshold value, we compute TPR and FPR, representing one point on the ROC curve.

In an ideal scenario, the curve would pass through the point (0,1), which represents value 1. for TPR and 0. for FPR. This means that for a given threshold value, the classifier is able to recognize all the positive data points (optimal benefits) and does not label any negative points as positive (0 cost). This scenario leads to an area under the curve (AUC) of 1. In general, measuring the area under the ROC gives a metric for the separation ability of the classifier. The said area oscillates between 0. and 1., with better results tending more towards 1. (as is the case with a perfect classifier).

For this task, I plotted the ROC and computed AUC for linear SVM (with soft and hard-margin), as well as RBF-kernel SVM. The plots and corresponding areas are shown in figure 1.

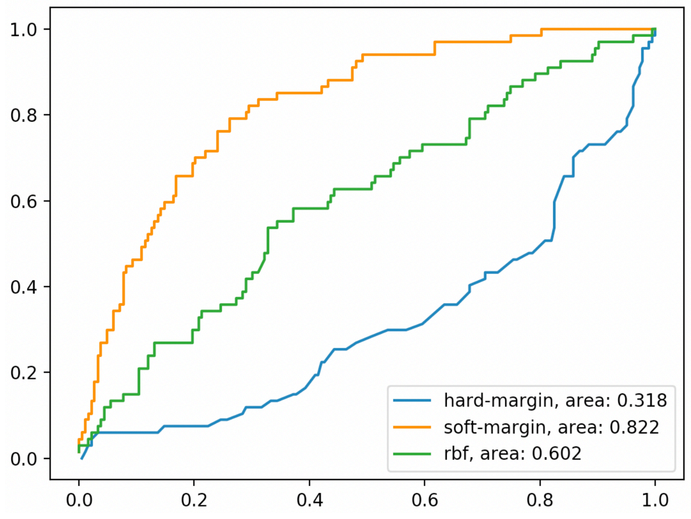

Figure 1 ROC-AUC for SVMs [hard-margin, soft-margin, RBF]

- **Part 3: Interpretation**

From the plots in figure 1, we can see that the linear SVM with soft-margin is the best performing classifier, since it has the largest AUC value (0.822). This means that the dataset we used is linearly-separable, but with noisy data. RBF-SVM has a curve close to the diagonal of the 2-D space, which means its results are barely better than random. This observation is further supported by the AUC value, which is of 0.6 (close to 0.5, which is the value of a random classifier). By inspecting the output of the RBF-SVM, we see that it predicts almost always negative values for all data points. It also returns a high accuracy for training data, and a low one for test data. This is usually a symptom of overfitting: since the RBF is flexible enough to capture many characteristics of a dataset, it can easily tune too much on the training set, leading to a poor generalization ability. Finally, the hard-margin linear classifier has an AUC-value of 0.3, which makes its predictions worse than a random draw. In theory, this means that the data points are too noisy, or contain too many outliers, to find a separating hyperplane with clean margins, and this is an expected result given the popularity of soft-margin SVMs with real datasets.

Task 2:

- **Task description:**

In this task, we would like to compare the performance of two variants of SVM classifiers (a soft-margin linear model and an RBF-kernel model) on the credits' dataset, and evaluate whether any perceived difference is due to random chance or to a difference in the classifiers' discrimination ability. This evaluation is done through hypothesis testing using the t-test. We chose accuracy as a measure of performance. We also evaluate the two models using two different resampling techniques: cross-validation and bootstrapped samples.

Part 1 presents briefly performance measures of the two models. Significance testing is discussed in part 2. Part 3 comments on the overall work and presents some ideas for further investigation.

- **Part 1: Performance measures and training techniques**

The task description proposes accuracy as a measure of performance of the classifiers. This is indeed a classic choice for classification tasks, as it captures the classifier's ability to distinguish between the two classes (accuracy is the ratio of correctly predicted labels over the total number of predictions). Other potential candidates include precision, recall and F1-score.

The first training scenario uses 10 rounds of 5-fold cross-validation. 5-fold cross validation divides the data set into 5 distinct sets and uses each of these sets as a test data, and the remaining 4 sets as training data in every iteration. We perform 5 such iterations to cover all the subsets. Usually, the overall accuracy of the model is estimated by combining accuracy results from each iteration, leading to an improved although still pessimistic error estimation. In this task, 5-fold cross validation is repeated 10 times, leading to 10 different data splits. At the end of the training, we would like to get a single accuracy value for each of the two models. To get this measure, we compute the mean accuracy after each cross-validation, then compute the mean of those means across all 10 iterations. We use the cross-validation means to calculate the variance of the accuracies and to perform a paired-t-test for statistical significance as explained in part 2. The code related to this task can be found in `cross_validation()` (*resources.py*, line 279). The mean and variance for both training scenarios are calculated and displayed by the function `summary_statistics()` (*resources.py*, line 231). The performance results for each model are shown in table 1.

Table 1 Mean and variance of accuracy in the cross-validation scenario

	Mean accuracy	Variance of CV accuracies
Linear SVM	0.749	1.54e-4
RBF-kernel SVM	0.797	3.42e-5

The second training technique is repeated bootstrapped samples. In this method, we sample a training set of the same size as the original dataset with replacement. Some data points in the training set would thus be repeated. We use the whole dataset as test data. Because of the large overlap between training and test data, the error/performance estimates we get using this method are highly optimistic, which can be seen by comparing the mean accuracies in the cross-validation and bootstrap scenarios. We repeat the procedure a number of times (in this task 50 times) to get a more accurate performance estimation. The code for this method can be found in `bootstrap()` (*resources.py*, line 306). The performance measurements for each model are shown in table 2.

Table 2 Mean and variance of accuracy in the bootstrap scenario

	Mean accuracy	Variance of CV accuracies
Linear SVM	0.762	1.36e-4
RBF-kernel SVM	0.890	5.87e-5

- **Part 2: Statistical significance using hypothesis testing**

In both training scenarios, we can see that the RBF-kernel performs better than the linear SVM. We would like to know if this perceived difference reflects a real gap in performance or is due to random chance/noisy data. We say that we want to know if the difference between the models' accuracies is statistically significant. We can answer this question using hypothesis testing [2]. Hypothesis testing uses test statistics to assess whether or not there is sufficient evidence to reject the null hypothesis. Each test uses a different test statistic, depending on the distribution of data points, the relationship between data samples, and other data specific factors. In this case, we set our hypothesis to be 'the difference in the performance of the two models is due to random chance', which is equivalent to 'there is no difference between the two accuracy means' [2]. We assume that our accuracy measures follow a student's t distribution, which is an alternative to the Gaussian distribution when we don't know the standard deviation of the population [3]. This allows us to use the t-test for hypothesis testing. Since both models were trained on the same data splits for each accuracy measures, we say that the data points are paired, which leads to the paired t-test for their evaluation [4].

The paired t-test computes a test-statistic called the t-score, which is a ratio of the difference of accuracy between models to the difference of accuracy within the models. Mathematically, this can be expressed as follows:

$$t = \frac{\Delta}{\frac{\sigma}{\sqrt{n}}}$$

Where: Δ is the average difference in the accuracies of the models, and $\frac{\sigma}{\sqrt{n}}$ is the estimated standard deviation of such differences.

A large t-score means that the models are more different between one another than they are within themselves. We thus need a large enough t-score to reject the null hypothesis and confirm the statistical significance of the difference between the two models [5]. To this end, the t-test proposes threshold values for t-scores given a certain probability value (α) and degrees of freedom. The probability value represents our confidence in the test results.

It measures exactly the likelihood of rejecting the null hypothesis. The smaller α the more confident we are in our rejection. Degrees of freedom is calculated as the input size – 1 (n-1 in our case). Traditionally, we would use the t-table to get the correct threshold for our scenario. Nowadays we can also use statistic libraries to get this estimate. I used scipy.stats.t.ppf() for that purpose.

If our t-score is less than the given threshold, we say that we don't have enough evidence to reject the null hypothesis, and conclude that the difference in performance is likely due to randomness in the data. If the t-score is above the threshold, we conclude with a 95% confidence the higher performing model is indeed a better classifier.

For both the cross-validation and the bootstrap scenario, I found that the difference in the classifiers' performance is statistically significant. The implementation of the test can be found in paired_t_test() the function computes the t-score manually using the formula mentioned above, and compares the result to the calculations of library functions from the scipy.stats package. The significance testing is also done using both t-values and p-values to prevent inconsistencies in the results.

- **Part 3: Notes on the t-test in model evaluation**

Recent literature points to flaws in the use of the t-test for model evaluation [2]. The flaws come from a violation of an assumption of the t-test regarding samples' independence. In both cross-validation and bootstrapping, training and test samples share data points, which makes them dependent on one another. I also noticed that the t-scores I get in both scenarios are quite high (absolute values of 8.204 for cross-validation and 59.63 in bootstrap sampling in a given run). I am not sure if such values are reasonable for this case. I tried to double check the implementation of the training procedures and they seem correct (or at least reflect my current understanding of the two methods). Another thing that is worth noting is that the t-test is typically used for small size data (less than 30 data points) and may not be suited for the bootstrap training procedure.

I read a little bit about alternative statistical tests but didn't get the chance to try any. I would like to explore McNemar's test and 2*5 cross-validation to double check the results of this test. I would also like to build a more fundamental understanding of the inner calculations of each of these tests to gain an intuition for why they may or may not work.

Task 3:

- **Task description**

We were asked to implement AdaBoost algorithm and apply it to linear SVM on credit dataset.

- **Part 1: AdaBoost and ensemble methods**

Ensemble methods in general aim to increase accuracy of predictions by combining the results of multiple classifiers. This can be done either by training the same model on different data (data-centered approach) or training different models/algorithms on the same data (model-centered approach). Boosting belongs to the former category. The aim of boosting in particular is to turn a weak classifier into a strong one (from high to low bias). The assumption here is that the poor performance of the classifier is due to an 'error' in its assumptions about the shape of the decision boundary. Generally, the higher the number of assumptions made, the simpler the model, and the higher its bias. In boosting, we adjust the bias of a weak model iteratively to account for misclassified data. The model generated in each iteration is better at classifying previously misclassified instances. At the end of the iterations, the models are able to cover many areas of the sample space, and therefore guarantee a higher classification accuracy (and reduced bias).

AdaBoost is one of the most famous boosting algorithms. It assigns weights to the data instances, initially set to a fixed value of 1/n where n is the number of data points. These weights increase subsequently for misclassified data points, and decreased for the correctly classified instances. The weights are either provided directly to the model if its algorithm supports their use (e.g., naïve Bayes classifiers would know how to handle such weights), or they're incorporated to the data through sampling. In the latter approach, data points are sampled with a probability proportional to their weights (instances with higher weights appear more often in the sample). The final predictions are a weighted sum of the predictions of each model. The weights for this final combination are set to be the learning rates (alpha) corresponding to each model. A pseudocode of the algorithm is shown in figure 2.

```
**Algorithm** *AdaBoost*(Data Set: $\mathcal{D}$, Base Classifier: $\mathcal{A}$, Maximum Rounds: $T$)
**begin**
  $t = 0$;
  **for** each $i$ initialize $W_1(i) = 1/n$;
  **repeat**
    $t = t + 1$;
    Determine weighted error rate $\epsilon_t$ on $\mathcal{D}$ when base algorithm $\mathcal{A}$
              is applied to weighted data set with weights $W_t(\cdot)$;
    $\alpha_t = \frac{1}{2}\log_e((1 - \epsilon_t)/\epsilon_t)$;
    **for** each misclassified $\overline{X_i} \in \mathcal{D}$ **do** $W_{t+1}(i) = W_t(i)e^{\alpha_t}$;
        **else** (correctly classified instance) **do** $W_{t+1}(i) = W_t(i)e^{-\alpha_t}$;
    **for** each instance $\overline{X_i}$ **do** normalize $W_{t+1}(i) = W_{t+1}(i)/[\sum_{j=1}^{n} W_{t+1}(j)]$;
  **until** $((t \geq T)$ OR $(\epsilon_t = 0)$ OR $(\epsilon_t \geq 0.5))$;
  Use ensemble components with weights $\alpha_t$ for test instance classification;
**end**
```

Figure 2 Pseudocode for AdaBoost (taken from textbook: Data Mining, The textbook by C. Aggarwal)

When using AdaBoost with linear SVM, I expect the ensemble model to perform better than the base (single) model. This is due to the fact that linear SVM is a simple model, with many assumptions on the decision boundary (linearity is usually a restraining assumption) and therefore a high bias. Boosting is also likely to help a little with variance, as shown in Task 4. To test this hypothesis, I evaluated a single linear SVM model in TASK3_4_BASE (*tasks.py*, line 58) using a normal data split approach. The following section will discuss the results of this evaluation.

- **Part 2: Empirical results**

The base accuracy of a single linear SVM is found in TASK3_4 and is usually between 0.722 and 0.744.

The implementation of the AdaBoost algorithm can be found in *tasks.py* under TASK3_4, starting at line 76. I used numpy's built-in function `numpy.random.choice()` to handle weighted sampling with replacement. The function supports sampling with a given set of probabilities. Said probabilities are calculated by normalizing the updated weights after every iteration (we divide the weights by the sum of all weights which puts their value between 0 and 1).

The accuracy of the ensemble method is around 0.776, sometimes going up to 0.789. We can see therefore that linear SVM indeed benefits from boosting.

When setting b to 0, we eliminate the weight update. In this case, the initial sampling of data points remains uniformly distributed, which leads to bootstrap sampling. AdaBoost is therefore changed to bagging.

Task 4:

- **Task description**

We are asked to compare the performance of boosting and bagging on both linear and RBF-kernel SVMs. I used the same code in TASK3_4 and TASK_3_4_BASE to apply bagging and boosting to both models as well as get the base results for comparison. Setting b=0 leads to bagging, and 0< b <1 gives AdaBoost algorithm as discussed earlier.

- **About bagging and boosting**

Bagging works on reducing the variance by combining models trained on different bootstrapped variations of the dataset. A 'bootstrap' is the result of sampling uniformly at random with replacement n elements from the dataset, where n is the size of the dataset. In other words, a bootstrap is a subset of the same size as the original dataset but with duplicate and sometimes missing original elements. Generating multiple such samples virtually increases the training data, and exposes the model to different data spaces at random. Combining different instances of the model leads to a more stable ensemble (i.e. with reduced variance).

Boosting, as explained earlier, focuses more on the bias, since it forces the trained model to give more attention to misclassified data points, resulting in an ensemble decision boundary better tuned to the data. When used with weighted sampling, boosting can reduce variance as well.

- **Empirical results**

The general observation is that boosting is better suited for linear SVM, while both boosting and bagging improve the performance of RBF-kernel SVM. Bagging does not work with linear SVM because the main issue of this classifier is bias and not variance. RBF-kernel benefits from both methods, and achieves a particularly high accuracy with boosting. Table 3 provides a summary of these empirical results. The reported values are those of accuracy.

Table 3 Accuracy values for linear and RBF SVM under with bagging and boosting

	Linear	RBF-Kernel
Base Model	0.732	0.668
Bagging	0.748	0.916
Boosting	0.784	1.0

Task 5:

The situation described in the task is a rare class learning scenario. In this situation, a target (usually positive) class is typically underrepresented in the data but generates a high cost when misclassified. Due to the scarcity of this class, a regular classifier might yield a high absolute accuracy by trivially predicting all classes as 'normal'. Basing the model training on maximizing the accuracy can thus result in costly misclassifications despite its high accuracy. The solution in this case is to find a way to incorporate misclassification costs in training the model.

There are two mains methods to account for misclassification costs: reweighting and resampling. In the first method, weights reflecting misclassification costs are associated with data instances. Classifying algorithms are then modified to take such weights into account. For many classifiers, like decision trees and naïve Bayes, including these weights is trivial as the model already works with weighted data. SVMs need a little more tuning since they do not naturally handle weighted data. The general approach is to add weights to the slack variables used in the soft-margin variant of the SVM. This pushes the decision boundary towards the normal class, which reduces the likelihood of misclassifying the rare class and increases that of misclassifying the normal class. If we don't want to modify the classifier's algorithm at all, we can use resampling. In this method, we either undersample the normal class or oversample the rare class. Undersampling has the advantage of resulting in higher training efficiency since the training set can be considerably small to account for the small number of rare classes. Another approach to resampling is to include all instances of the rare class with some instances of the normal class in the training set.

This last approach is the one I implemented in TASK5 (*tasks.py*, line 12). I then tested both linear SVM and RBF-kernel SVM against a randomly sampled test data. For both models, resampling lead to a decrease in the false negative ratio on the test set. Linear model went from 0.5 to 0.3 in a random run, while the RBF-kernel model returned consistently a false negative rate of 0.

No collaborators.

Sources

[1] Wikipedia contributors. (2019, November 30). Receiver operating characteristic. In *Wikipedia, The Free Encyclopedia*. Retrieved 16:53, January 5, 2020, from https://en.wikipedia.org/w/index.php?title=Receiver_operating_characteristic&oldid=928547878

[2] Brownlee, J. (2019, August 18). Statistical Significance Tests for Comparing Machine Learning Algorithms. Retrieved January 26, 2020, from https://machinelearningmastery.com/statistical-significance-tests-for-comparing-machine-learning-algorithms/

[3] Wikipedia contributors. (2020, January 21). Student's t-test. In Wikipedia, The Free Encyclopedia. Retrieved 21:19, January 26, 2020, from https://en.wikipedia.org/w/index.php?title=Student%27s_t-test&oldid=936900480

[4] T Test (Student's T-Test): Definition and Examples. (n.d.). Retrieved January 26, 2020, from https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/t-test/

[5] Aankul, A. (2017, August 30). T-test using Python and Numpy. Retrieved January 26, 2020, from https://towardsdatascience.com/inferential-statistics-series-t-test-using-numpy-2718f8f9bf2f