Project 3 Report : Compare classifiers in scikit-learn library

Sajal Kumar

Implementation details on parameters

My implementation uses all 6 methods on their default setting except the following general parameters that could be changed:

- `random_state` : Random seed (set to 1 by default).
- `max_iter` : maximum number of iterations for methods using gradient descent (set to 10 by default)
- `n_jobs` : Number of parallel threads allowed (set to 4 by default).

My implementation also allows changes to the following classifier specific parameters:

- Decision Tree
  - `min_samples_split` denoted by `dt_min_split` (set to 10 by default).
- Linear Support Vector Machine
  - `penalty` denoted by `lsvm_penalty` (set to ‘l2’ by default).
  - `C` denoted by `lsvm_c` (set to 0.05 by default).
- Non Linear Support Vector Machine
  - `C` denoted by `nlsvm_c` (set to 0.05 by default).
  - `gamma` denoted by `nlsvm_gma` (set to ’auto’ by default).
- Perceptron
  - `penalty` denoted by `ptron_penalty` (set to ‘l2’ by default).
  - `alpha` denoted by `ptron_c` (set to 0.05 by default).
  - `eta0` denoted by `ptron_eta` (set to 0.001 by default).

- Logistic Regression
  - `penalty` denoted by `logres_penalty` (set to '12' by default).
  - `C` denoted by `logres_c` (set to 0.05 by default).
- KNN classifier
  - `n_neighbors` denoted by `knn_k` (set to 3 by default).
  - `algorithm` denoted by `knn_algo` (set to 'kd-tree' by default).

Apart from the above mentioned parameter some other classifier parameters were also changed but were not provided as a parameter for the user:

- Decision Tree
  - `min_samples_leaf` was set to 5 (commonly used in significance testing).
- Linear Support Vector Machine
  - `fit_intercept` set to 'False' (since the data was properly scaled and centralized).
- Perceptron
  - `fit_intercept` set to 'False' (since the data was properly scaled and centralized).
- Logistic Regression
  - `fit_intercept` set to 'False' (since the data was properly scaled and centralized).
  - `solver` set to 'sag' (faster version of stochastic gradient descent).
  - `multi_class` set to 'multinomial' (Reason explained later).

Hence the above represents the 'standard' setting for all classifier when no parameter is changed. Next I will present results on 'Digits' and 'REALDISP Activitiy Recognition' data-sets using the 'standard' setting and then discuss and show the difference in results on the 'Digits' data-set with alternate configuration.

Performance on Digits data-set

Table 1 shows the runtime (in seconds), accuracy on testing data (in %) and accuracy on training (in %) for 'Digits' data-set using the 'standard' configuration on 6 classifiers Dec. Tree (Decision Tree), Lin. SVM (Linear SVM), N-lin. SVM (Non-Linear SVM), Perc. (Perceptron), Logi. Reg. (Logistic Regression) and KNN class. (KNN classification). The data-set was scaled using the 'StandardScaler' method from sklearn. 70% of the data was randomly partitioned for training and the rest 30% was used for testing. Stratified partitioning was used. Logistic regression and KNN classifiers work very well followed by Non-Linear SVM, Linear SVM and Decision Tree. Perceptron performed the worst.

Info	Dec. Tree	Lin. SVM	N-lin. SVM	Perc.	Logi. Reg.	KNN class.
runtime	0.011	0.036	0.021	0.15	0.10	0.004
acc. test	83	88	92	72	95	98
acc. train	92	92	94	75	97	99

Table 1: Result on 'Digits' data-set with 'standard' configuration of 6 classifiers.

Performance on REALDISP data-set

Since REALDISP is a huge data-set wherein evaluating all log files is very impractical on a personal laptop (and the project description does not mandate the utilization of the entire REALDISP dataset), I decided to use the 'ideal' log files of 4 subjects (3, 4, 6 and 7). Since this data-set is still huge (more than 500,000 samples), a large `max_iter` would be bad. Thus, we used the 'standard' `max_iter` = 10 for this analysis. Surely, Linear SVM, Non-Linear SVM, Perceptron and Logistic Regression would suffer because of that but according to the results, Linear SVM and Perceptron take the biggest hit. This data-set is a true test of run-time.

Info	Dec. Tree	Lin. SVM	N-lin. SVM	Perc.	Logi. Reg.	KNN class.
runtime	216	62	32	12	32	4
acc. test	98	28	65	37	81	99
acc. train	99	28	65	37	81	99

Table 2: Result on 'REALDISP' data-set with 'standard' configuration of 6 classifiers.

Table 2 shows the runtime (in seconds), accuracy on testing data (in %) and accuracy on training (in %) for 'Digits' data-set using the 'standard' configuration on 6 classifiers Dec. Tree (Decision Tree), Lin. SVM (Linear SVM), N-lin. SVM (Non-Linear SVM), Perc. (Perceptron), Logi. Reg. (Logistic Regression) and KNN class. (KNN classification). The data-set was scaled using the 'StandardScaler' method from sklearn. 70% of the data was randomly partitioned for training and the rest 30% was used for testing. Stratified partitioning was used. Decision Tree and KNN classifiers work very well followed by Logistic Regression and Non-Linear SVM. Linear SVM and Perceptron performed the worst. It clearly seems like KNN classifier is very powerful, being fast and effective.

A more comprehensive evaluation of classifiers

In this section we would discuss and show changes in performance when certain parameters were tweaked. We are using 'Digits' data-set. We only tweaked parameters for those classifier that did not perform well, that is, they reported an accuracy ≤ 90, in 'standard' configuration. Thus, we only considered Perceptron, Linear SVM and Decision Tree classifiers.

- Decision Tree

– `min_samples_split` : The following table shows the change in results at 4 different values of `min_samples_split` The results in Table 3 are expected, increasing

values →	**10**	**40**	**50**	**100**
runtime	0.011	0.010	0.010	0.009
acc. test	83	80	80	77
acc. train	92	86	86	79

Table 3: Change in Decision tree result with changing `min_samples_split`

the `min_samples_split` decreases the runtime (slightly) and degrades the performance because the pre-pruning might be happening pre-maturely, leading to under-fitting.

- Linear Support Vector Machine

  – `C` : The following table shows the change in results at 4 different values of `C`, we do not show runtime as no visible changes were noticed. The results in Table 4 are

values →	**0.05**	**0.01**	**1**
acc. test	88	91	83
acc. train	92	93	87

Table 4: Change in Linear SVM result with changing `C`

expected, increasing `C` decreases accuracy as we are moving towards under-fitting, whereas reducing it improves accuracy.

- Perceptron Perceptron's bad performance could've been due to low number of max iterations and thus, I increased that number to 50 and sure enough the performance improved. The next set of results all have `max_iter` set to 50.

  – `penalty` : The following table shows the change in results at 2 different values of `penalty`, we do not show runtime as no visible changes were noticed. The results

values →	**l2**	**l1**
acc. test	81	32
acc. train	81	32

Table 5: Change in Perceptron results with changing `penalty`

in Table 5 are interesting as I would have expected similar (or slightly worse) performance between the two penalty scheme, however, a big drop in performance can be seen.

values →	0.05	0.01	1
acc. test	81	77	82
acc. train	81	81	82

Table 6: Change in Perceptron results with changing `alpha`

- `alpha` : The following table shows the change in results at 3 different values of `alpha`, we do not show runtime as no visible changes were noticed. 'l2' penalty term was being used. The results in Table 6 are again surprising as lowering the `alpha` degrades the result while increasing it improves the result.
- `eta0` : The following table shows the change in results at 3 different values of `eta0`, we do not show runtime as no visible changes were noticed. `penalty` was set to 'l2' and `alpha` to 1. The results in Table 7 are expected, increasing `eta0`

values →	0.001	0.01	0.0001
acc. test	81	77	79
acc. train	81	78	81

Table 7: Change in Perceptron results with changing `eta0`

degrades performance while increasing it also degrades performance and thus 0.01 seems to be a good value.

Additionally, setting `multi_class` as 'ovr' (one versus rest) in logistic regression made it very slow for the REALDISP data-set. It makes sense as the there are 34 class-labels in the REALDISP data-set which means the 34 models (with more than 500,000 samples) have to be made. The reason for Logistic regression's slow behavior is the fact that it uses stochastic gradient descent that works on samples, making it very slow. Thus, we set the `multi_class` to 'multinomial' for Logistic regression and it worked fine.

I also tried to see what happens in I increased the 'K' for KNN classifier and the results were expected, increasing the 'K' from 3 to 10 degraded the performance on test data from 98 to 97. Increasing 'K' to 20 degraded the performance on test data from 98 to 96. This result was inline with what Dr. Cao mentioned in the class about smaller 'K' performing better.

Understanding pruning strategies in DecisionTreeClassifier

DecisionTreeClassifier is an optimized implementation of the CART algorithm. Sklearn currently does not support any post-pruning strategies. However, it provides several options for pre-pruning in the form of parameters, some of which are:

- `max_depth` : The maximum depth of the tree.

- `min_samples_split` : The minimum number of samples required to split an internal node.
- `min_samples_leaf` : The minimum number of samples required to be at a leaf node.
- `min_impurity_decrease` : A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
- `min_weight_fraction_leaf` : The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node.

In the *sklearn-Github* repository for tree classification – `https://github.com/scikit-learn/scikit-learn/tree/` :

- `max_depth` is utilized in `_tree.pyx` at line number 223 where, if the current depth exceeds texttmax_depth then the node is annotated as a leaf node.
- `min_samples_split` is utilized in `_tree.pyx` at line number 224 where, if the number of samples at the current node is less than texttmax_depth then the node is annotated as a leaf node.
- `min_samples_leaf` is utilized in `_tree.pyx` at line number 225 where, if the number of samples at the current node is less than $2*$ `min_samples_leaf` then the node is annotated as a leaf node.
- `min_impurity_decrease` is utilized at several places in `_tree.pyx`, one instance is at line number 241 where, if the improvement gained by splitting the current node, added with `EPSILON` (machine limits for floating point types), is less than `min_impurity_decrease` then the node is annotated as a leaf node.
- `min_weight_fraction_leaf` is first utilized in `tree.py` at line number 272 (or 275 depending on the condition) to compute `min_weight_leaf` = `min_weight_fraction_leaf` $*n\_samples$ (or `min_weight_fraction_leaf` $*sum(sample\_weight)$). `min_weight_leaf` is then used at line 226 in `_tree.pyx` where, if the sum of the weighted samples is less than $2*$ `min_weight_leaf` then the node is annotated as a leaf node. Weighting the samples is a specially useful when the class-labels are unbalanced in which case `min_weight_fraction_leaf` will make the pre-pruning less biased toward dominant classes.