Perceptrons and Support-Vector Machines

Filippo Chiandotto[1], Federico Andres Dotti[2], and Simone Tosato[3]

[1] Corso di laurea magistrale in Scienze Statistiche, matricola 1234567
filippo.chiandotto@studenti.unipd.it
[2] Corso di laurea magistrale in Scienze Statistiche, matricola 1202662
federicoandres.dotti@studenti.unipd.it
[3] Corso di laurea magistrale in Scienze Statistiche, matricola 1202709
simone.tosato@studenti.unipd.it

Abstract. The work presented in this report has the goal of exploring two supervised binary classification techniques: the perceptron and the support-vector machine. We started off with the base algorithms as they are presented in the textbook and then went on to explore variations of these in order to enhance their scalability to high-dimensional data sets. For the task at hand, we chose to use Python together with the `numpy` library, which provides more efficient data structures and matrix algebra operations. In order to verify their efficiency we then applied these algorithms to real data.

Keywords: SVM · Perceptron · MapReduce · Center Distance Ratio Method · Support Vectors · EBR

1 Introduction

In machine learning, the perceptron and the support-vector machine are some of the most widely used solutions to deal with the problem of binary classification. There's a myriad of problems that can be modeled in such a manner. Was a certain transaction fraudulent? Is an email spam? Does a specific song belong to the jazz genre? Any such question can be answered with a binary classifier. In general, the efficiency of a binary classification algorithm depends on the specific data we want to classify. The perceptron is ideal when the space of input variables (or *feature space*) is linearly separable. A feature space is linearly separable when there exists a hyperplane that perfectly separates the two classes. Under linear separability the SVM also performs well, but the SVM is advantageous compared to the perceptron when the data are not perfectly separable, because it is capable of dealing with misclassified data through a penalization system.

We will be dealing with data in matrix form. That is, $X_{n\times p}$ will be our data matrix where n is the number of observations and p is the number of features associated to each observation. There is also a label vector $y \in \{-1, +1\}^n$ that indicates which class each observation belongs to. In practice, the data is often presented as a matrix $[X|y]_{n\times(p+1)}$ where the vector of labels is concatenated

to the data matrix. We will use the following notation: $\mathbf{x}_i$ is the p-dimensional feature vector corresponding to the i-th observation, $\mathbf{x}_{ij}$ corresponds to the j-th element of $\mathbf{x}_i$, y_i and y_i' are the class and the predicted class for the i-th observation.

2 The Basics

2.1 Perceptron

The perceptron is a linear binary classifier. Its input is a vector $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{ip}) \in \mathbb{R}^p$. Perceptrons can be thought of as functions $f : \mathbb{R}^p \times \mathbb{R}^p \mapsto \{-1, +1\}$ of a weights vector $\mathbf{w} \in \mathbb{R}^p$ and an input vector (or *feature vector*), containing data, such that:

$$f(\mathbf{x}_i, \mathbf{w}) = \begin{cases} +1 & \text{if } \langle \mathbf{x}_i, \mathbf{w} \rangle > \theta \\ -1 & \text{if } \langle \mathbf{x}_i, \mathbf{w} \rangle \leq \theta \end{cases}$$

Here $\langle , \rangle$ is the standard inner product. To train a perceptron we need to find one vector of weights $\mathbf{w}$ such that the observations with $y = +1$ are on the positive side of the hyperplane (that is, $f = +1$) and for the observations with $y = -1$ we have that $f = -1$. First we show an implementation of the perceptron with $\theta = 0$ and then we extend it to an arbitrary θ with a simple tweak.

Algorithm 1 Perceptron with $\theta = 0$

```
Initialize w ∈ R^p randomly
Repeat until convergence
    for (x_i, y_i) ∈ [X|y] do
        y_i' ← ⟨x_i, w⟩
        if y_i' and y_i have the same sign then
            t is properly classified
        else
            w ← w + η y_i x_i
        end if
    end for
```

Here $t = (\mathbf{x}_i, y)$ is a training example and η is an arbitrary learning rate that we choose. Algorithm 1 is to be applied several times to the training set until onvergence is achieved [3].. The algorithm is stopped once one of these conditions

is met:

- Every point is correctly classified.
- The proportion of correctly classified observations on the current iteration is the same as the previous one.
- An arbitrary number of iterations has been reached.

The way to allow for a variable threshold is to add another component to the feature vectors as well as the weights vector. If for a 0 threshold algorithm we have a vector of weights $\mathbf{w} = (w_1, ..., w_p)$, we proceed by replacing it by $\mathbf{w'} = (w_1, ..., w_p, \theta)$ and replace every feature vector $\mathbf{x}_i = (x_{i1}, ..., x_{ip})$ by $\mathbf{x'}_i = (x_{i1}, ..., x_{ip}, -1)$. By applying Algorithm 1 to the new training set defined by $\mathbf{x'}_i$ we obtain the desired solution. This follows because $\langle \mathbf{x'}_i, \mathbf{w'} \rangle > 0$ is equivalent to $(\sum_{k=1}^{p} w_n x_{ik}) - \theta = \langle \mathbf{x}_i, \mathbf{w} \rangle - \theta > 0 \iff \langle \mathbf{x}_i, \mathbf{w} \rangle > \theta$.

An issue with perceptrons is that when the data is linearly separable there are infinitely many hyperplanes that can correctly classify our observations. The perceptron will converge to one of these hyperplanes depending on the initialization of $\mathbf{w}$, but there is no notion of whether there could be a more desirable conclusion or not. A solution to this problem is obtained through a more complex algorithm known as the support-vector machine, which will be introduced next.

2.2 SVM

With the support-vector machine, the goal is to maximize the margin γ between the hyperplane and any of the points in the training set. The hyperplane is given by $\langle \mathbf{x}, \mathbf{w} \rangle + b = 0$ where b is the so-called bias. Formally, given a training set $(\mathbf{x}_i, y_i), i = 1, ..., n.$ we want to maximize γ subject to $y_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) > \gamma$ and we will use a normalized vector $\frac{\mathbf{w}}{\|\mathbf{w}\|}$ for the hyperplane rather than $\mathbf{w}$. Why maximize γ? Note that as the observations get closer to the plane, the inner product $\langle \mathbf{x}, \mathbf{w} \rangle + b$ gets closer to 0, thus maximizing this product over the entire training set will lead to a hyperplane that pushes the observations as far as possible. This solution is called the optimal hyperplane. We will also require that the parallel hyperplanes where the support vectors lie are described by the equations $\langle \mathbf{x}_i, \mathbf{w} \rangle + b = +1$ and $\langle \mathbf{x}_i, \mathbf{w} \rangle + b = -1$. By projecting the support-vectors on the negative hyperplane onto the positive hyperplane we find the relationship $\gamma \frac{\langle \mathbf{w}, \mathbf{w} \rangle}{\|\mathbf{w}\|} = 1$. Thus we can see that maximizing the margin is equivalent to minimizing the norm of the weights vector because $\gamma = \frac{1}{\|\mathbf{w}\|}$.

Finding the Optimal Hyperplane In theory, we would like the support vectors to be the points closer to the hyperplane. In practice, especially in the context of big data, there will be many data points between the two parallel hyperplanes due to the data not being separable. Mathematically, if the $i - th$ observation is of this type it will follow that $\|\langle \mathbf{x}_i, \mathbf{w} \rangle + b\| < 1$. The idea is to construct a loss function (or penalty function) that discourages us to use

a solution where the points are too close to the hyperplane, that is, we want the penalty to be proportional to how close our observation is to the separating hyperplane $\langle \mathbf{x}, \mathbf{w} \rangle + b = 0$. We now have the two components to define a function we are to minimize to find the optimal hyperplane. One of the conditions is that we have to minimize $\|\mathbf{w}\|$. Notice that this is equivalent to minimizing $\frac{\|\mathbf{w}\|^2}{2}$ because $\|\mathbf{w}_1\| < \|\mathbf{w}_2\| \implies \frac{\|\mathbf{w}_1\|^2}{2} < \frac{\|\mathbf{w}_2\|^2}{2} \forall \mathbf{w}_1, \mathbf{w}_2 \in \mathbb{R}^p$. This function is convenient because $\frac{\partial}{\partial w_i} \frac{\|\mathbf{w}\|^2}{2} = w_i$. We will build a function that combines $\frac{\|\mathbf{w}\|^2}{2}$ together with the penalty and minimize it. A commonly used penalty function is

$$f_1(\mathbf{w}, b) = C \sum_{i=1}^{n} \max\{0, 1 - y_i(\sum_{j=1}^{p} w_j x_{ij} + b)\}$$

An important piece of the penalty function is the loss function (also known as *hinge function*)

$$L(\mathbf{x}_i, y_i) = \max\{0, 1 - y_i(\sum_{j=1}^{p} w_j x_{ij} + b)\}$$

Notice that when $\|\langle \mathbf{x}_i, \mathbf{w} \rangle + b\| \geq 1$ it follows that $L(\mathbf{x}_i, y_i) = 0$. This means that the penalty will be 0 if our point $\mathbf{x}_i$ is far from the hyperplane, otherwise, the penalty will be proportional to how close it is to the hyperplane by a factor of C, an arbitrary regularization parameter that we choose. The larger C is the higher the incurred penalty will be. Note that $1 - y_i(\sum_{j=1}^{p} w_j x_{ij} + b)$ increases linearly as $\mathbf{x}_i$ approaches the separating hyperplane.

The derivative of the loss function is also important since we will need it to find the minimum. It follows that

$$\frac{\partial L}{\partial w_j} = \begin{cases} -y_i x_{ij} & \text{if } y_i(\sum_{j=1}^{p} w_j x_{ij} + b) < 1 \\ 0 & \text{otherwise} \end{cases}$$

We can now put both pieces together to obtain the function that we'll have to minimize:

$$f(\mathbf{w}, b) = \frac{1}{2} \sum_{i=1}^{p} w_j^2 + C(\sum_{i=1}^{n} \max\{0, 1 - y_i(\sum_{j=1}^{p} w_j x_{ij} + b)\} \tag{1}$$

Summing the derivatives, we get that

$$\frac{\partial f}{\partial w_j} = \begin{cases} w_j + C(\sum_{i=1}^{n} -y_i x_{ij}) & \text{if } y_i(\sum_{j=1}^{p} w_j x_{ij} + b) < 1 \\ w_j & \text{otherwise} \end{cases} \tag{2}$$

Gradient Descent We can use gradient descent to find f's minimum. Since the gradient $\nabla f(\mathbf{w}, b)|_{(\mathbf{w},b)=(\mathbf{w}_1,b_1)}$ is the direction of steepest ascent at point $(\mathbf{w}_1, b_1)$, the direction of greatest descent at the same point is equal to $-\nabla f(\mathbf{w}, b)|_{(\mathbf{w},b)=(\mathbf{w}_1,b_1)}$. The idea is to descend in this direction until finding a

minimum. Notice that for the bias b we can do the same trick we did with the threshold θ in the perceptron, and take the bias as the $(p+1)-th$ element of the weights vector.

Algorithm 2 Gradient Descent

Set values for C, η, and initialize $\mathbf{w} \in \mathbb{R}^{p+1}$ randomly
Repeat until convergence:
  for $(x_i, y_i) \in [X|y]$ **do**
    Compute $\frac{\partial f}{\partial w_j}$ $\forall j = 1, ..., p+1$
    Set $w_j \leftarrow w_j - \eta \frac{\partial f}{\partial w_j}$
  end for

Once the weights $\mathbf{w}$ get close after repeating Algorithm 2, we have converged to a minimum of f. Convergence is achieved when one of these conditions is met:

- There has been a certain number of iterations without a change in the function's minimum value.
- An arbitrary number of iterations has been reached.

Stochastic Gradient Descent If the number of training samples is too large, then using gradient descent may take too long because for each data point, when we are updating the values of the parameters, we are using the complete training set. A slight variation of gradient descent consists in uniformly choosing a training example at random and computing the gradient only on this observation. This algorithm is known as stochastic gradient descent. Since we are computing the gradient of f only on one observation, the sum $C(\sum_{i=1}^{n} -y_i x_{ij})$ has only one term, so equation (2) is simplified as follows:

$$\frac{\partial f}{\partial w_j} = \begin{cases} w_j - C y_i x_{ij} & \text{if } y_i(\sum_{j=1}^{p} w_j x_{ij} + b) < 1 \\ w_j & \text{otherwise} \end{cases} \tag{3}$$

Algorithm 3 Stochastic Gradient Descent

Set values for C, η, and initialize $\mathbf{w} \in \mathbb{R}^{p+1}$ randomly
Repeat until convergence:
  Shuffle the rows of $[X|y]$
  for $(x_i, y_i) \in [X|y]$ **do**
    Compute $\frac{\partial f}{\partial w_j}$ $\forall j = 1, ..., p+1$
    Set $w_j \leftarrow w_j - \eta \frac{\partial f}{\partial w_j}$
  end for

Note that it is normal that some elements of the training set are never used in this variation of the algorithm. Although stochastic gradient descent minimizes the loss function faster, it is noisier and the solution generally oscillates around the the minimum, giving some variation in accuracy. A common problem with all types of gradient descent is caused by the presence of local minima in the function we are trying to minimize. The existence of these points might cause the algorithm to stop prematurely and returning a solution of $\mathbf{w}$ that is not corresponding to the global minimum we are looking for.

3 Scope of the Project

The algorithms presented in section 2 constitute the base. Even if the conditions are met for them to converge to a suitable solution, the size of the input data and consequently the size of the training set can be too large. This may lead to rapidly increasing execution times and usage of memory. To handle this problem, we will discuss several methods which in most cases can be used simultaneously. These methods belong to one of two classes. The first class consists of exploiting the mathematical properties of the data in order to reduce their size to a more manageable magnitude. Among these methods we will consider the center distance ratio method (CDRM) [2] and we will introduce a new epoch based reduction of the data. The second class is that of distributing the execution of the algorithm. We will implement the SVM and the perceptron with `mrjob`, a Python package developed by Yelp to facilitate the implementation of MapReduce [4].

4 Methods

4.1 Training set Reduction

Center Distance Ratio Method (CDRM) When training a SVM it can be observed that the separating hyperplane will be adjusted based on whether the training observation has been misclassified, weighted by its distance to the plane. If an unclassified observation is close to its own class and far to the opposite class, then it is reasonable to assume that it will be correctly classified, therefore it is a sound approach to ignore such observations. To formalize these thoughts we have to define the center of each class. Let $d(\mathbf{x}_i, \mathbf{x}_j)$ be the standard Euclidean distance, that is: $d(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{p} \sqrt{(\mathbf{x}_{ik} - \mathbf{x}_{jk})^2}$ and let $\{\mathbf{x}_1, ..., \mathbf{x}_{n^+}\}^+$ be the training examples belonging to the positive class and $\{\mathbf{x}_1, ..., \mathbf{x}_{n^-}\}^-$ the training examples that belong to the negative class. Let $m_{x^+} = \frac{1}{n^+} \sum_{i=1}^{n^+} \mathbf{x}_i$ be the center for the positive class and $m_{x^-} = \frac{1}{n^-} \sum_{i=1}^{n^-} \mathbf{x}_i$ be the center for the negative class. Note that n^+ and n^- denote the number of positive and negative training example respectively and that $n^+ < n$, $n^- < n$. For each element $\mathbf{x}_i$ in the positive class compute the ratio between the distance to the center of its own class and the negative class $R_{x^+} = \frac{d(\mathbf{x}_i, m_{x^+})}{d(\mathbf{x}_i, m_{x^-})}$. Do the analogous procedure with

the negative examples to obtain $R_{x^-} = \frac{d(\mathbf{x}_i, m_{x-})}{d(\mathbf{x}_i, m_{x+})}$. If the corresponding ratio is smaller than an arbitrary threshold of choice, then we get rid of that training example. The reduced training set will be the union of the reduced positive class and the reduced negative class:

$$\{\mathbf{x}_i \in \{\mathbf{x}_1, ..., \mathbf{x}_{n^+}\}^+ : R_{x^+} < \theta\} \cup \{\mathbf{x}_i \in \{\mathbf{x}_1, ..., \mathbf{x}_{n^-}\}^- : R_{x^-} < \theta\}$$

This approach is not viable when working with categorical data given the fact that the Euclidean distance is not suitable for non-numerical values.

EBR: Epoch Based Reduction Consider the usual training set X with which we want to train a SVM and let SV be the set of support vectors associated with the optimal hyperplane. It follows that adapting a SVM on X yields the exact same solution as adapting a SVM on SV. In this case none of the observations will be discarded, and the set of support vectors will be the same as the original input set. This is of interest to us because adapting a SVM on SV is much less costly than doing so on the entire training set. We will modify stochastic gradient descent by iteratively restricting the training set of the $k-th$ epoch to the set of potential support vectors in the $(k-1)-th$ epoch plus the misclassified observations, which will result in a substantial runtime reduction. By the set of potential support vectors we mean the points that are close to the positive and negative hyperplanes, $\langle \mathbf{x}, \mathbf{w} \rangle + b = 1$ and $\langle \mathbf{x}, \mathbf{w} \rangle + b = -1$, respectively. Let $M^{(t)}$ be the set of misclassified observations at the $t-th$ epoch, we define the set PS of potential support vectors as

$$PS = \{\mathbf{x}_i \in X : |\langle \mathbf{x}, \mathbf{w} \rangle + b| \leq 1\}$$

Algorithm 4 Epoch Based Reduction

Set values for C, η, and initialize $\mathbf{w} \in \mathbb{R}^{p+1}$ randomly
Set $t = 0$ and $[X|y]^{(0)} = [X|y]$
Repeat until convergence: ▷ $[X|y]^{(t)}$ is the reduced training set at the $t-th$ epoch
  Shuffle the rows of $[X|y]^{(t)}$
  for $(x_i, y_i) \in [X|y]^{(t)}$ **do**
    Compute $\frac{\partial f}{\partial w_j}$ $\forall j = 1, ..., p+1$
    Set $w_j \leftarrow w_j - \eta \frac{\partial f}{\partial w_j}$
  end for
  Set $X^{(t+1)} \leftarrow PS^{(t)} \cup M^{(t)}$
  Set $t \leftarrow t + 1$

When implementing Algorithm 4 it is often the case that too many points are ignored and that the rate at which we reduce the training set in each epoch is too high, especially in the first few epochs, leading to a solution that does not coincide with the optimal hyperplane. This happens due to the fact that the

definition of potential support vector is too restrictive. We want to tweak the idea and be more conservative with the reduction of our training set in order to arrive at the best solution. Consider the inequality $|\langle \mathbf{x}, \mathbf{w} \rangle + b| \leq c$ for some positive $c \in \mathbb{R}$ and notice that, as c increases, it becomes more likely that an arbitrary training example $\mathbf{x}_i$ satisfies that condition. We will redefine the set of potential support vectors to enjoy less abrupt reductions in the training set, as follows:

$$PS' = \{\mathbf{x}_i \in X : |\langle \mathbf{x}, \mathbf{w} \rangle + b| \leq 1 + a_k\}$$

Here a_k is a monotone decreasing sequence in $\mathbb{R}$ that converges to 0, so in the limit PS and PS' coincide.

4.2 MapReduce

Both the perceptron and the SVM can be distributed in order to make them more scalable. In this section we present a MapReduce implementation for both of these algorithms.

Perceptron To implement MapReduce for the perceptron, we first divide the dataset in chunks. Each node must know the current value of the weights vector $\mathbf{w}$. For each training example in the chunk we compute $y_i' = \langle \mathbf{x}_i, \mathbf{w} \rangle$. If $\mathbf{x}_i$ is misclassified then the mapper produces the key $(j, \eta y_i \mathbf{x}_{ij})$ if $\mathbf{x}_{ij} \neq 0$. If it is equal to 0 then it won't affect the successive iteration of the weights vector thus we don't need to produce a key for this specific component. The reduce function takes each key j and adds the associated products $\eta y_i \mathbf{x}_{ij}$ which are in turn added to $\mathbf{w}_j$, the j-th component of $\mathbf{w}_j$.

Algorithm 5 Perceptron With MapReduce

```
Method Mapper(w, x_i, η, y_i)
y_i' ← ⟨x_i, w⟩
if y_i ≠ y_i' then
    for j = 1, ..., p + 1 do
        if x_ij ≠ 0 then
            Yield key-value (j, x_ij η y_j)
        end if
    end for
end if

Method Reducer(w, j, x_ij η y_j)
Yield(j, w_j + Σ_{i=1}^{n} x_ij η y_j)
```

This implementation can deal with sparse data too. Since we compute $\eta x_{ij} y_j$ only when $x_{ij} \neq 0$ we will simply ignore all the missing values that won't give any contribution to the update of w_j.

SVM Like we did with the perceptron, we first have to divide our data in chunks in order to implement MapReduce. We assume that we have L nodes working simultaneously and we start with an empty set SV of global support vectors. The first step is to compute SVM in each chunk, and for each node l we save the associated support vectors which we will call SV_l. We then add all the support vectors of each chunk to the set of global support vectors SV and that concludes the first iteration. We then modify each chunk by adding the set of global support vectors. Notice that after each iteration the chunks grow in size by the same amount, namely the number of global support vectors that were not in the chunk beforehand. Since this version of MapReduce modifies the chunks at each epoch we will use $C_l^{(t)}$ to denote the chunk of node l at time t where $C_l^{(0)}$ is the original chunk.

Algorithm 6 SVM With MapReduce

Method Mapper(t, SV)
Yield key-value $(l, C_l^{(t)})$ where $C_l^{(t)} = C_l^{(t-1)} \cup SV$

Method Reducer
Train SVM on $C_l^{(t)}$ to obtain SV_l, the support vectors at node l
Yield (l, SV_l)

We iterate over the mapper and the reducer until convergence: as every chunk gets more similar to the others they will start "agreeing" on which points of the training set are the real support vectors for the whole dataset. The algorithm stops when the global support vectors of two consecutive iterations are equal[1].

5 Experiments

Our experiments were based on UC Irvine's HIGGS data set that presents a classification problem to distinguish between a signal process that produces Higgs bosons and a background process which does not [5]. The data consist of 11.000.000 observations which have been produced with Monte Carlo simulations. Each observation has 28 features. The first 21 are kinematic properties measured by the particle detectors in the accelerator and the remaining 7 are *high level features*, which are functions of the first 21 features and can aid the classification process. In order to test the performance of our algorithms we compared their execution times and the accuracy using samples of different sizes from the original dataset. It was not possible to test the MapReduce version of the two methods because we lack a computer cluster to run the algorithms with.

Figure 1 shows the execution time in seconds as a function of the size of the data in logarithmic scale with base 10. The first thing to notice is that there is

a clear distinction between SVM and its various modifications and the perceptron. The graph makes it seems as the perceptron has a constant runtime, but this is due to the SVM being more expensive computationally. We can also see that EBR greatly reduces the runtime, with it being comparable to that of the perceptron for training sets up to approximately $10^{4.5}$ observations. For training sets larger than $10^{4.25}$ observations it can be seen that the runtime of regular SVM decreases. This might be due to stochastic gradient descent finding a local minimum rather than the optimal solution of interest. The same can be said about SVM with CDRM on training sets larger than 10^5 observations.

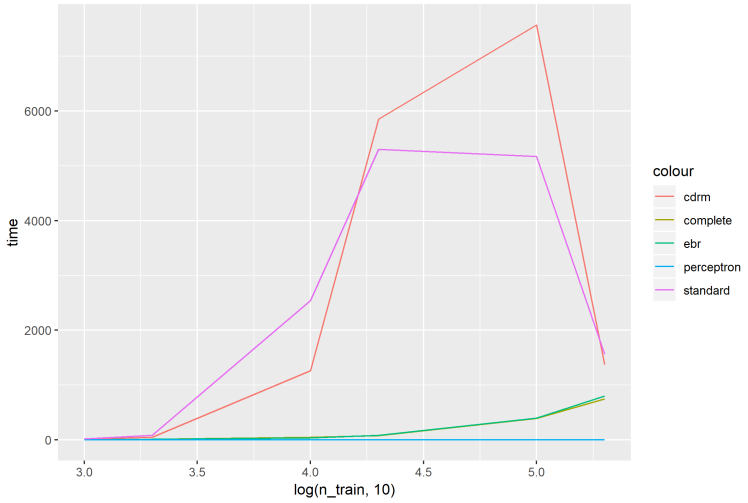

Fig. 1: Execution time

Figure 2 shows the accuracy in percentages as a function of the training set's size. Notice that the perceptron is clearly inferior to SVM and its modifications, but as seen in the previous figure it is also much faster. In this case there is a trade-off between the desired accuracy and the time it will take to train an algorithm. For the smallest training set size we experimented on the regular SVM fared much better. This is due to the other methods removing sensitive points, as these methods get rid of a significant part of the training set. As the size of the training set grows larger than $10^{4.25}$ all of the SVM-based methods tend to converge to the same accuracy, which suggests that the fastest method should be chosen if dealing with big training sets.

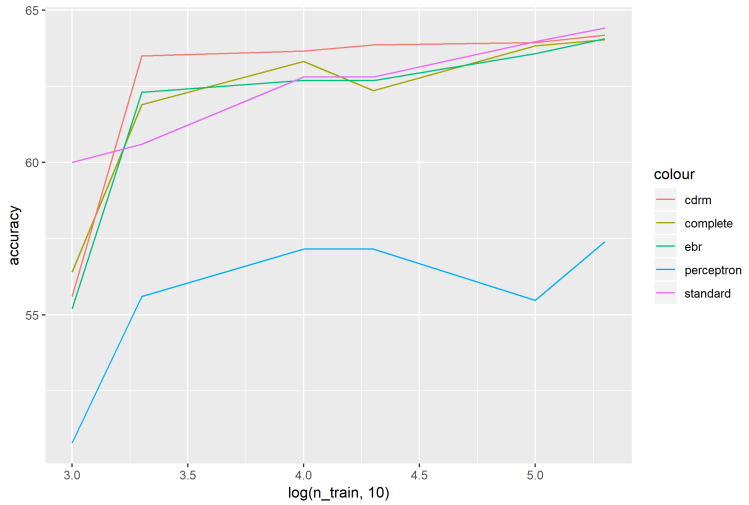

Fig. 2: Accuracy

In binary classification, the recall is defined as the ratio $\frac{TP}{TP+FN}$, where TP are the true positives and FN are the false negatives. The precision is defined as the ratio $\frac{TP}{TP+FP}$ where FP are the false positives. A small recall implies that we are overclassifying observations in the negative class, while a small precision indicates that most of the points classified as positive were incorrectly classified as such. Figure 3 shows the F1 score, which is the harmonic average of the precision and the recall, as a function of the size of the training set. In our experiments, the F1 score and the accuracy present similar behaviors, with the exception that the F1 score is nearly the same in all non-standard SVM methods.

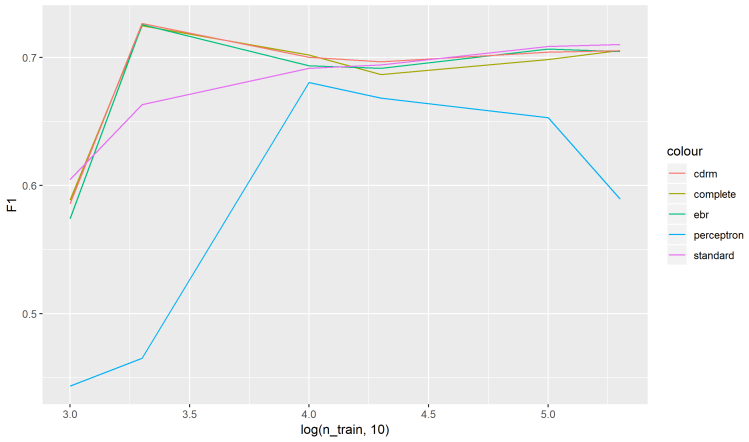

Fig. 3: F1

References

1. Catak, Ferhat Ozgur and Erdal Balaban, Mehmet: A MapReduce-based distributed SVM algorithm for binary classification. Turkish Journal of Electrical Engineering and Computer Sciences, 2013. doi: 10.3906/elk-1302-68
2. Li Zhang, Weida Zhou and Licheng Jiao, Pre-extracting Support Vectors for Support Vector Machine, Proceedings of 5th International Conference on Signal Processing, Beijing, pp. 1427-1431, August, 2000.
3. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of massive datasets, chapter 12. Cambridge university press (2014)
4. mrjob v0.5.10 documentation, https://pythonhosted.org/mrjob/.
5. UCI Machine Learning Repository: HIGGS Data Set, https://archive.ics.uci.edu/ml/datasets/HIGGS.


(a) Filippo Chiandotto


(b) Federico Andres Dotti


(c) Simone Tosato