**WOLF**

[machine learning **WO**rkf**L**ow management **F**ramework]

A report submitted to the EECS department of

the University of Kansas

**Advisors:** Dr. Luke Huan, Dr. Fengjun Li, Dr. Bo Luo

**By:** Pranav Shekhar Bahl

December, 2016

**1. Introduction**

Recently machine learning has been creating great strides in many areas of science like health, finance, education, sports etc. which has encouraged demand for machine learning systems amongst novices in the field. Though by definition machine learning automates the task of learning in terms of rule induction, classification, regression etc. which is then used to draw knowledgeable insights and also to forecast an event before it actually takes place but despite that automation it doesn’t automate the task of selecting the best algorithm(s) for a specific dataset. With the rapidly growing machine learning algorithms it becomes difficult for novices as well as researchers to choose the best algorithm. The crux of a machine learning system is to solve fundamental problem of preprocessing the data to help machine learning algorithm understand the data better, also to solve the problem of choosing meaningful features hence reducing the noise from the data and finally choosing the best resulting machine learning algorithm which is performed by doing grid search over hyperparameters of various machine learning algorithms and afterwards doing metric comparison amongst all outcomes. This is the problem we address in the work.

Automation is the fuel that drives WOLF. Automating time-consuming and repeatable tasks are the defining characteristics of the project. The rising scope of Artificial Intelligence (AI) and machine learning increases the need for automation to simplify the process and hence help researchers and data scientists to dig deeper into the problem and understanding it well than spending time in tweaking the algorithms. The positive correlation of growing intelligence and the complexity of solutions has shifted the trend from Artificial Intelligence (AI) to Automated Intelligence, a paradigm on which WOLF is based.

WOLF has been built to have impact on wider audience, the automation of machine learning pipeline helps people with different level of expertise and requirement, helping novices to identify best combination of algorithms without even having in depth knowledge of available algorithms and helping the researchers and businesses with better machine learning knowledge to figure out best resulting hyperparameters. The project is developed with the idea behind rising interest of data scientists towards online data science competitions the machine learning automation has become a necessity as competitors spend ~40% of the time in model construction and evaluation.

Though the project so far supports the classification problems but there are handful of classification algorithms consisting of base classifiers and ensemble methods as well. With comparably low availability of algorithm selection project provides the leverage of adding your own algorithm to a transaction making it more customizable and the scope of growing algorithms in the system is unrestricted. The process of adding a algorithm to the pipeline is pretty intuitive and straight forward. Also the idea of not stealing the freedom of model evaluation from the user has been taken care of in the project, so result file consists of various widely used metric calculations giving user better understanding and easy comparison of pipeline combinations and hence helping user to make decisions with high confidence.

**2. Literature Survey**

The growth in demand and shortage of data scientists has lately been causing a transformation from Artificial Intelligence (AI) to automation. Automating the machine learning tasks has been the part of game from early 90’s, during that time the automation wasn’t use to be that aggressive and usually use to automate an algorithms like a software product launched by Unica­­­­­ in 1995 with a sole purpose of tuning parameters of neural nets followed by an automated modeling solution by MarketSwitch in 1998.

In the last several years, the leading analytic software vendors (SAS and IBM) have also added automated modeling features to their products. SAS provided with techniques like missing value treatment, outlier detection, feature selection and data modeling. It takes user dataset and response measure as input. IBM product provided the same capabilities but in different way.

All the software product mentioned above are the proprietary products. Reflecting the machine learning community’s transition towards open source and it’s not surprising that most widely used and innovative machine learning projects now comes from open source community. Some of the worth mentioning products are Auto-WEKA, TPOT and Data Robot.

Auto-WEKA was developed by four researchers at the University of British Columbia and Freiburg University. Auto-WEKA consists of classification algorithms only. The software conducts search across 39 different algorithms consisting of various basic classifiers and ensemble methods. Since the parameter space of multiple algorithms is huge so it use Bayesian approach to solve the problem.

TPOT has been developed by Randal S. Olson at Computational Genetics Lab. It was first released with support to automation of decision trees and random forest with some feature preprocessing but currently it supports all different kinds of algorithm for regression as well as classification. It is a python library which can be installed on any compatible system. TPOT takes user data file as input and follows genetic programming paradigm to search for best solution amongst various possibilities in the pipeline.

DataRobot is an automated data science solution developed by some of the top Kagglers (participants at Kaggle competitions). It is a complete proprietary data science system for the businesses interested in getting insights of their data and also interested in solving data problems. The software covers all the tasks which a data scientist performs, might be even beyond that. It automates the task of data cleaning, visualization, model construction, model evaluation and making predictions.

WOLF on the other hand is composed of various machine learning tasks ranging from data pre-processing to selecting the best model for user provided data. The uniqueness of the software comes by making user the owner of product as they can control each transaction of pipeline if required by configuring it. A user can restrict the number/type of transactions, can run the models of his/her choice, specify the range of hyperparameters for each algorithm and can even be the decision maker by figuring out the best model for his/her data using the results omitted by the software. Result file consists of metric evaluations and their visualizations portraying the comparison of different model combinations. WOLF is built with an idea of giving user a leverage of customizing the product. A user can even add more algorithms to the pipeline following the WOLF protocol. By providing flexibility and customization, software has become easily adaptable; hence useful for wider audience.

**3. WOLF Architecture**

WOLF works in transactions to figure out the best pipeline for the users data. A transaction is nothing but a common machine learning task performed to build the best solution for a data problem. Some of those tasks include data pre-processing, cross validation, feature engineering, model construction and evaluation. Each transaction provides choice of various algorithms; based on users provided configuration the project runs all transactions and select a pipeline consisting of best performing algorithm for each transaction individually. Each transaction in the project is isolated from one another but is dependent on the output generated by its prior task. WOLF’s workflow looks something like:

**WOLF Automation**

Model Construction

Feature Selection

Splitting data

Feature Extraction

User Data

WOLF Configuration

-

Model Selection

Pre-processing

Metric Evaluation

A typical pipeline of WOLF consists of multiple transactions, not all transactions are mandatory. The dotted line in above figure is to show alternate paths or the transactions that can skipped in a pipeline. Each transaction performs a meaningful task.

1. **Pre-processing**: There are some common data pre-processing operations which are performed on the data file provided by user. This step includes operations like getting rid of NaN (Not a Number) values or replacing them with user specified value. Label encoding of categorical features and scaling of numerical features also coms under pre-processing step.
2. **Splitting data:** The splitting data transaction replicates the task of cross validation. After pre-processing of user input data it is then split into multiple pairs of train and test files. The number of generated train and test data files combinations depend on the users provided values for the number of folds and repetitions parameter under split data configuration. The data file is always randomized before splitting which leads to distinct pairs of train and test files.
3. **Feature Extraction:** The data extraction is mostly meant for dimensionality reduction of data set or creating new features by combining multiple features which might end up being more meaningful to model construction algorithms. The train and test files combinations created by splitting data transaction is treated as input for data extraction. Because not all the data sets requires data extraction or even if user want to take control of the feature extraction, this transaction has been made optional.
4. **Feature Selection:** Taking the train and test files pairs generated after data extraction or splitting data transaction (if data extraction is skipped) the operation of feature selection is performed on them. The idea behind this transaction is to get rid of noisy data or the features which are not meaningful to model construction algorithms. The task of feature selection is also optional for the same reasons as feature extraction. After performing feature selection pair of train and test files are created but this time the output files consist of meaningful features only.
5. **Model Construction:** The input for model construction transaction is train and test files combinations generated by feature selection or feature extraction or splitting data transaction based on the users configuration if any of the transaction is skipped or not. This transaction consists of multiple algorithms and each model construction algorithm runs in parallel. The output generated by this transaction consists of result files which contains true values and predictions made for test files. The number of generated files depend on the number of algorithms chosen by user for experimentation and also the freedom of hyperparameters as configured by users.
6. **Model Evaluation:** After performing all the important tasks of machine learning, evaluations of results is undertaken. In this step metric calculation is performed for all possible combinations formed by prior transactions. WOLF consists of metrics like precision, accuracy, area under curve (AUC), Matthews correlation coefficient (MCC), f1-score and many more. The output of this transaction is used for decision making or choosing the best (based on the chosen metrics) predicting combination of algorithms.
7. **Model Selection:** This is the moment of truth, the user is provided with a result file consisting of metrics calculated for all possible combination based on configuration file. The result file also consists of graph representation of all metrics for each configuration combination which is then used by user to choose best combination. This gives users a leverage of analyzing the result and choose the algorithm which performs best according to their metric selection.

A sample WOLF pipeline looks something like:

**WOLF Automation**

User Data

**CV**

-Folds 5

-Repition 3

PCA

SVM RFE

Random Forest

Metric Evaluation

Model Selection

WOLF Configuration

- fill Na(0)

- LE features

In this sample pipeline WOLF takes users data file and configuration file as input. Configuration file here consists of parameters for each transaction of WOLF. In this case following operations happen:

1. Pre-process data by filling NA values with 0 and label encoding the categorical features, if any
2. Split data in 5 folds and repeat the process of splitting for 3 times. After execution of this step the output will be 15 pairs of train and test set
3. Taking 15 pairs of train and test set generated in previous step, WOLF will run Principal Component Analysis (PCA) algorithm for feature extraction. After this process new 15 pairs of train and test set will be generated.
4. The newly generated train and test pairs after feature extraction is taken as input for feature selection. In this step Support Vector Machine – Recursive Feature Elimination (SVM-RFE) algorithm is executed for all pairs. The output of this step will again be 15 pairs of train and test set but with meaningful features only.
5. At this step two different model construction algorithms (Random Forest and SVM) will run in parallel. Once the model training has been performed on training data, predictions will be made for testing data which is then saved for metric evaluation. The result files consists of true labels as well as predictions.
6. Once we have results/predictions after model construction, metric evaluation step is performed to calculate various metrics. At the moment WOLF will calculate MCC, F1-score, AUC-ROC, precision, accuracy, and recall for the results.
7. All the results generated in previous step are saved in database to figure out the best performing solution/pipeline. In this step a result file is generated by collaborating results for individual model construction algorithms. The file also consists of some visualization of metrics calculated in previous step to make model selection easier for a user.

**4. WOLF Database Management**

In WOLF, the configuration of all the transactions get saved. And also the metric evaluation results of each configuration get saved into the database. The idea behind integrating database to the project is to make the tracking of each run easier. It also helps to ease the process of result generation after the complete pipeline completion which is then utilized for model selection. As the configuration of each user differs from each other so using relational database could have caused huge waste of space. To fill the gap of space wastage a non-relational database, MongoDB is used in which only the parameters for a transaction that are specified by users get saved. The database schema of the project looks like:

* \_id (PK)
* parameters
* data\_file

Split Data

* \_id (PK)
* train\_file
* test\_file

Files

* \_id (PK)
* executable
* parameters

Feature Extraction

* \_id (PK)
* executable
* parameters

Feature Selection

* \_id (PK)
* executable
* parameters

Algorithm

Result

Pre-Processing

* \_id (PK)
* data\_file
* parameters

­

* \_id (PK)
* pp\_id(FK)
* file\_id(FK)
* sd\_id(FK)
* feature\_extraction\_id(FK)
* feature\_selection\_id(FK)
* algo\_id(FK)
* f1\_score
* mcc
* precision
* roc\_auc
* recall
* standard\_deviation
* accuracy
* model\_path
* feature\_importance\_path

The use-case of each collection is:

1. **Split Data:** The user’s configuration for “Splitting Data” transaction of the project gets saved into this collection. The collections consists of data file information by storing the input file name and also the parameters like number of folds and number of repetitions set by user in the configuration file.
2. **Files:** This collection keeps track of the files name that get created after splitting data transaction. It just consists of train file name and test file name for each pair generated. The idea behind this is to keep track of best performing pair if required.
3. **Feature Extraction:**  The configuration passed for “Feature Extraction” transaction by user gets saved into this collection. The information stored in the collection consists of, executable (name of algorithm that get executed) and parameters for the chosen executable.
4. **Feature Selection:** The feature selection collection provides information about the users configuration for “Feature Selection” transaction. It consists of executable (name of algorithm that get executed) information and also consists of parameters values provided for chosen algorithm.
5. **Algorithm:** The information related to the “Model Construction” transaction gets stored into this collection. The stored information consists of executable (name of user selected algorithm) and parameters for the selected algorithm.
6. **Result:** As the name explains this collection contains output generated by “Model Evaluation” transaction. All the metrics value that get calculated in evaluation step like mcc, f1-score, precision, and recall etc. gets stored into the collection. The result collection provides moment of truth; it is used to query the results of all pipeline combinations executed by user and generate the report for model selection.

**\*\***Each document in all the collections has unique id

**Database Relations:**

All the collections in WOLF database schema except Result has one to many relationship with Result collection. As Result collection contains the metric evaluation result so it makes sense to refer each result document to the configuration it is related to. The Result collection along with metric evaluation output, consists of unique ids’ for all the other collections like split data, feature extraction & selection, algorithm. The unique ids’ of other collections are nothing but the foreign keys that point to the particular configuration used for the generated result.

**5. The Protocol**

The WOLF transactions are dependent on the user input configuration. To keep the consistency across all user configurations, the project follows a protocol which defines the structure and keywords used for creating configuration file. Following is the screenshot of a sample configuration file.

­­­

Every part of the configuration has meaning bind to the task to be performed. The protocol is divided into three parts and following are the meaning attached to them.

1. **Transaction names**: WOLF consists of five user configurable transactions. Each name has a keyword with a meaning assigned to it.
2. preprocessing – Pre-processing keyword as the name say is used to define data pre-processing configuration
3. datasplit – This keyword is used to specify split data configuration
4. feature\_extraction – This is the keyword used to represent feature extraction configurations
5. feature\_selection – The configuration of feature selection is defined under this tag
6. algorithm – The algorithm keyword characterize the model construction configuration
7. metric\_calculation – This keyword defines the metric evaluation configuration
8. **Transaction Variable:** In WOLF each transaction has their reserved keywords knows as parameters. These parameters makes the project customizable by restricting the tasks based on users preference. Following are the available customizable parameters under each transaction:
9. executable – This keyword is used to specify executable file name for a transaction/algorithm
10. data\_file – To specify the path of data file this keyword is used and it is only available under ‘datasplit’ configuration
11. output\_folder – As the name signifies this keyword specifies the path of output folder where all the WOLF results and logs get saved
12. parameters – This is used to define values for customizable parameters for an algorithm under a transaction.
13. missing\_values – This is used to specify the value used to fill Not a Number (NaN) values in dataset
14. label\_encoding – This keyword is used to figure out if user wants categorical features to be label encoded or not
15. no\_of\_files – This keyword is used to split the execution commands into multiple files for better interoperability. This option is available under all the transactions as WOLF creates shell script for each transaction individually.

1. **Parameter Value:** The value of a particular parameter under a transaction can be specified in two different ways:
2. **single:** This keyword is used to signify the discrete value of a variable
3. **collection:** This keyword implies that there are more than one value assigned to a parameter and WOLF runs the particular transaction for each specified value. This is further divided into two parts:
4. **list:** This keyword is used if user want to specify multiple values for a given parameter
5. **range:** As the name signifies this keyword is used if user wants a transaction to be run for different values of a parameter defined in terms of range.

**6. Future Work**

The WOLF in the near future will have some other major features coming up. Also a user gets leverage to expand WOLF as per their use. The future work of the project is divided into two categories.

1. **Web interface:** One of the majorupcoming feature of WOLF is to design a web interface. The idea behind designing a user interactive interface is to abstraction to the project and also ease the process of creating WOLF configuration. The web interface will provide easy pick and execute platform. After the completion of pipeline execution it will also display the results in term of easily interpretable visualizations.
2. **WOLF extension:** WOLF gives user a leverage to implement any algorithm and make it part of the pipeline execution. The new algorithm can be easily added to the system by following the WOLF protocol to make sure it complies to the rules. For easy integration a user can refer to the available template.

**APPENDIX**

**Table of Contents**

1. [**Classifiers**](#Classifier)
   1. [**Adaboost**](#adaboost)
   2. [**Naïve Bayes**](#BernoulliNaïveBayes)
      1. [**Bernoulli Naïve Bayes**](#BernoulliNaïveBayes)
      2. [**Gaussian Naïve Bayes**](#GaussianNaïveBayes)
   3. [**Decision Tree**](#DecisionTree)
   4. [**Logistic Regression**](#LogisticRegression)
   5. [**Random Forest**](#RandomForest)
   6. [**SVM**](#SVM)
      1. [**C-SVM**](#CSVM)
      2. [**Linear SVM**](#LinearSVM)
      3. [**NU SVM**](#NuSVM)
   7. [**Linear Discriminant Analysis**](#LinearDiscriminantAnalysis)
   8. [**Quadratic Discriminant Analysis**](#QuadraticDiscriminantAnalysis)
2. [**Feature Selection Method**](#Featureselectionmethod)
   1. [**SVM RFE**](#SVMRFE)
3. [**Dimensionality Reduction**](#DimensionalityReduction)
   1. [**Principle Component Analysis**](#PrinciplecomponentAnalysis)

**1.** **Classifier**

**1.1 Adaboost****: AdaBoostClassifier.py**

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. This class implements the algorithm known as AdaBoost-SAMME.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| e | base\_estimator –  The base estimator from which the boosted ensemble is built. | object | DecisionTreeClassifier |  |
| n | n\_estimator –  The maximum number of estimators at which boosting is terminated. | int | 50 | In case of perfect fit, the learning procedure is stopped early. |
| r | learning\_rate –  Learning rate shrinks the contribution of each classifier by learning\_rate. There is a trade-off between learning\_rate and n\_estimators. | float | 1.0 |  |
| a | Algorithm |  | ‘SAMME.R' | If ‘SAMME.R’ then use the SAMME.R real boosting algorithm. base\_estimator must support calculation of class probabilities.  If ‘SAMME’ then use the SAMME discrete boosting algorithm. |

**1.2 Naïve Bayes**

**1.2.1** **Bernoulli Naïve Bayes: BernoulliNaiveBayes.py**

Naïve Bayes Classifiersare a family of simple probabilistic classifiersbased on applying Bayes' theoremwith strong (naive) independence assumptions between the features.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| Alpha | Alpha –  Additive (Laplace/Lidstone) smoothing parameter | Float | 1.0 | 0 for no smoothing |

**1.2.2** **Gaussian Naïve Bayes: GaussianNaiveBayes.py**

**1.3** **Decision Tree: DecissionTree.py**

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| C | Criterion - The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain | string | “gini” | “gini”, “entropy” |
| f | max\_features - The number of features to consider when looking for the best split | int, float, string or None | None | * If int, then consider max\_features features at each split. * If float, then max\_features is a percentage and int(max\_features \* n\_features) features are considered at each split. * If “auto”, then max\_features=sqrt(n\_features). * If “sqrt”, then max\_features=sqrt(n\_features). * If “log2”, then max\_features=log2(n\_features). * If None, then max\_features=n\_features. |
| d | max\_depth - The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples | int or None | None | If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples |
| s | min\_samples\_split - The minimum number of samples required to split an internal node | int or float | 2 | * If int, then consider min\_samples\_split as the minimum number. * If float, then min\_samples\_split is a percentage and ceil(min\_samples\_split \* n\_samples)are the minimum number of samples for each split |
| l | min\_samples\_leaf - The minimum number of samples required to be at a leaf node | int or float | 1 | * If int, then consider min\_samples\_leaf as the minimum number. * If float, then min\_samples\_leaf is a percentage and ceil(min\_samples\_leaf \* n\_samples)are the minimum number of samples for each node |
| w | min\_weight\_fraction\_leaf - The minimum weighted fraction of the input samples required to be at a leaf node | float | 0. | >= 0. |
| n | max\_leaf\_nodes - Grow a tree with max leaf nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes | int or None | None | If None then unlimited number of leaf nodes |
| p | presort - Whether to presort the data to speed up the finding of best splits in fitting | bool | False | For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training. |

**1.4** **Logistic Regression: logestic\_regression.py**

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the ‘multi\_class’ option is set to ‘ovr’, and uses the cross- entropy loss if the ‘multi\_class’ option is set to ‘multinomial’. (Currently the ‘multinomial’ option is supported only by the ‘lbfgs’, ‘sag’ and ‘newton-cg’ solvers.)

This class implements regularized logistic regression using the ‘liblinear’ library, ‘newton-cg’, ‘sag’ and ‘lbfgs’ solvers. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The ‘newton-cg’, ‘sag’, and ‘lbfgs’ solvers support only L2 regularization with primal formulation. The ‘liblinear’ solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| C | C –  Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization. | Float | 1.0 | any positive float |
| p | Penalty –  Used to specify the norm used in the penalization. | string | “l2” | “l1”, “l2” |

**1.5** **Random Forest: RandomForest.py**

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| T | The number of trees in the forest | int | 10 | >=5 |
| D | Max\_depth –The maximum depth of the tree. | int | None | If None, then nodes are expanded until all leaves are pure or until all leaves contain less than 2 samples |
| C | Criterion - The function to measure the quality of a split | string | “gini” | Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain |
| S | **min\_samples\_split -** The minimum number of samples required to split an internal node | Int or float | 2 | * - If int, then consider min\_samples\_split as the minimum number. * - If float, then min\_samples\_split is a percentage and ceil(min\_samples\_split \* n\_samples) are the minimum number of samples for each split. |
| L | **min\_samples\_leaf -** The minimum number of samples required to be at a leaf node | Int or float | 1 | * - If int, then consider min\_samples\_leaf as the minimum number. * - If float, then min\_samples\_leaf is a percentage and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node. |
| M | **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 | float | 0 | Samples have equal weight when sample\_weight is not provided. |
| N | **max\_leaf\_nodes -** Grow trees with max\_leaf\_nodes in best-first fashion | int or None | None | Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. |
| I | **min\_impurity\_split -** Threshold for early stopping in tree growth | float | 1e-7 | A node will split if its impurity is above the threshold, otherwise it is a leaf. |
| W | **warm\_start – signifies whether to reuse the solution of the previous call or not** | bool | False | When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. |

**1.6** **SVM**

**1.6.1** **C-SVM: CSupportVectorClassification.py**

The implementation is based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

The multiclass support is handled according to a one-vs-one scheme.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| C | C –  Penalty parameter C of the error term | Float | 1.0 | Any positive float number |
| K | kernel –  The loss function | string | “rbf” | Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to precompute the kernel matrix. |
| D | Degree –  Degree of the polynomial kernel function (‘poly’) | int | 3 | Degree of the polynomial kernel function (‘poly’). Ignored by all other kernels. |
| G | Gamma – Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. | Float | ‘auto’ | If gamma is ‘auto’ then 1/n\_features will be used instead. |
| f | Coef0 –  Independent term in kernel function. | float | 0.0 | It is only significant in ‘poly’ and ‘sigmoid’. |
| S | Shrinking – Whether to use the shrinking heuristic | bool | True | True, False |
| P | Probability – Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method. | bool | False | True, False |
| T | Tol – Tolerance for stopping criterion | float | 0.001 |  |
| A | Cache\_size –  The size of kernel cache(in MB) | float | 200 |  |
| w | Class\_weight | string | None | If not given, all classes are supposed to have weight one.  “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* number of occurrences of each value in y) |
| v | Verbose | bool | False | this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context. |
| i | Max\_iter – The maximum number of iterations to be run. | int | -1 | Hard limit on iterations within solver, or -1 for no limit. |
| e | Decision\_function\_shape | String | None | ‘ovo’: the original one-vs-one (‘ovo’) decision function of libsvm which has shape (n\_samples, n\_classes \* (n\_classes - 1) / 2)  ‘ovr’: Whether to return a one-vs-rest (‘ovr’) decision function of shape (n\_samples, n\_classes) as all other classifiers |
| r | Random\_state –  The seed of the pseudo random number generator to use when shuffling the data. | Int or None | None | None |

**1.6.2** **Linear SVM: LinearSupportVectorClassification.py**

Similar to C-SVM with parameter kernel=’linear’, but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| C | C –  Penalty parameter C of the error term | Float | 1.0 | Any positive float number |
| P | Penalty –  The norm used in the penalization | string | “l2” | “l1”, “l2” |
| L | Loss –  The loss function | string | “squared\_hinge” | ‘hinge’ is the standard SVM loss while ‘squared\_hinge’ is the square of the hinge loss. |
| d | Dual –  Select the algorithm to either solve the dual or primal optimization problem | bool | True | Prefer dual=False when n\_samples > n\_features |
| t | Tol –  Tolerance for stopping criteria | Float | 1e-4 | Any float number |
| m | Multi\_class –  Determines the multi-class strategy if y contains more than two classes | string | “ovr” | If ‘ovr’, it trains n\_classes one-vs-rest classifiers.  If ‘crammer\_singer’, it optimizes a joint objective over all classes and the options loss, penalty and dual will be ignored. |
| f | Fit\_intercept –  Whether to calculate the intercept for this model | bool | True | If set to false, no intercept will be used in calculations |
| s | Intercept\_scaling | Float | 1 | When self.fit\_intercept is True, instance vector x becomes [X, Intercept\_scaling]. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept\_scaling has to be increased. |
| w | Class\_weight | string | None | If not given, all classes are supposed to have weight one.  “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* number of occurrences of each value in y) |
| v | Verbose | int | 0 | this setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context. |
| i | Max\_iter –  The maximum number of iterations to be run. | int | 1000 | Positive integer. |
| r | Random\_state –  The seed of the pseudo random number generator to use when shuffling the data. | Int or None | None | None |

**1.6.3** **Nu SVM: NuSupportVectorClassification.py**

Similar to C-SVM but uses a parameter to control the number of support vectors.

The implementation is based on libsvm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| N | Nu –  An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. | float | 0.5 | Should be in the interval (0, 1]. By default 0.5 will be taken. |
| K | kernel –  The loss function | string | “rbf” | Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. If none is given, ‘rbf’ will be used. If a callable is given it is used to precompute the kernel matrix. |
| D | Degree –  Degree of the polynomial kernel function (‘poly’) | int | 3 | Degree of the polynomial kernel function (‘poly’). Ignored by all other kernels. |
| G | Gamma – Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. | Float | ‘auto’ | If gamma is ‘auto’ then 1/n\_features will be used instead. |
| F | Coef0 –  Independent term in kernel function. | float | 0.0 | It is only significant in ‘poly’ and ‘sigmoid’. |
| S | Shrinking – Whether to use the shrinking heuristic | bool | True | True, False |
| P | Probability – Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method. | bool | False | True, False |
| T | Tol – Tolerance for stopping criterion | float | 0.001 |  |
| A | Cache\_size –  The size of kernel cache(in MB) | float | 200 |  |
| W | Class\_weight | string | None | If not given, all classes are supposed to have weight one.  “balanced” mode uses the values of y toaautomatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* number of occurrences of each value in y) |
| V | Verbose | bool | False | this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context. |
| I | Max\_iter – The maximum number of iterations to be run. | int | -1 | Hard limit on iterations within solver, or -1 for no limit. |
| E | Decision\_function\_shape | String | None | ‘ovo’: the original one-vs-one (‘ovo’) decision function of libsvm which has shape (n\_samples, n\_classes \* (n\_classes - 1) / 2)  ‘ovr’: Whether to return a one-vs-rest (‘ovr’) decision function of shape (n\_samples, n\_classes) as all other classifiers |
| R | Random\_state –  The seed of the pseudo random number generator to use when shuffling the data for probability estimation. | int seed | None | Int seed. |

**1.7** **Linear Discriminant Analysis: LinearDiscriminantAnalysis.py**

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes’ rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| S | Solver | string | ‘svd’ | ‘svd’: Singular value decomposition (default). Does not compute the covariance matrix, therefore this solver is recommended for data with a large number of features.  ‘lsqr’: Least squares solution, can be combined with shrinkage.  ‘eigen’: Eigenvalue decomposition, can be combined with shrinkage. |
| V | shrinkage | String or float | None | None: no shrinkage (default).   * ‘auto’: automatic shrinkage using the Ledoit-Wolf lemma. * float between 0 and 1: fixed shrinkage parameter. |
| N | n\_components | int | None | Number of components (< n\_classes - 1) for dimensionality reduction. |
| T | tol | float | 0.0001 | Any float number |

**1.8 Quadratic Discriminant Analysis: QuadraticDisriminatAnalysis.py**

A classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes’ rule.

The model fits a Gaussian density to each class.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| P | reg\_param | float | 0.0 | Regularizes the covariance estimate as  (1-reg\_param)\*Sigma + reg\_param\*np.eye(n\_features) |

**2.** **Feature selection method**

**2.1** **SVM\_RFE: SVM\_RFE.py**

Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and weights are assigned to each one of them. Then, features whose absolute weights are the smallest are pruned from the current set features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| N | The number of features | Int | None | The number of features to select. If None, half of the features are selected |
| S | steps | Int or float | 1 | If greater than or equal to 1, then step corresponds to the (integer) number of features to remove at each iteration. If within (0.0, 1.0), then step corresponds to the percentage (rounded down) of features to remove at each iteration. |

**3.** **Dimensionality Reduction**

**3.1** **Principle Component Analysis: PCA.py**

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter\_name | Description | Value Type | Default value | Possible values/conditions |
| n | n\_component –  Number of components to keep | Int or float or string | None | if n\_components is not set all components are kept: N\_components = min(n\_samples, n\_features)  if n\_components == ‘mle’ and svd\_solver == ‘full’, Minka’s MLE is used to guess the dimension if 0 < n\_components < 1 and svd\_solver == ‘full’, select the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by n\_components n\_components cannot be equal to n\_features for svd\_solver == ‘arpack’. |
| c | copy | Bool | True | * If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit\_transform(X) instead. |
| w | whiten | Bool | False | When True (False by default) the components\_ vectors are multiplied by the square root of n\_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances. |