Eötvös Loránd University
Faculty of Informatics

Investigating the Importance of Meta-features for Classification Tasks in Meta-learning

Dr. Tomáš Horváth
Head of Data Science Department at ELTE

Jehad M. Aldahdooh
Computer Science

Budapest, 2017.

Acknowledgement

I would like to express my gratitude to my supervisor Dr. Tomáš Horváth. His guidance and motivation helped me a lot from the first day of this research.

I would also like to thank my university -Eötvös Loránd University- for giving me this chance to study at such a great university.

Last but not least, I would like to thank my family and teachers from my home country especially to Dr. Aiman Abu Samra for supporting me in my life in Budapest.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis outline 2

2 Machine learning **1**

2.1 Binary classification Models 1

2.1.1 K-Nearest Neighbors 1

2.1.2 Logistic regression 4

2.1.3 Support vector machine: 8

2.1.4 Decision Trees 11

2.1.5 Random Forest: 14

2.1.6 Extreme Gradient Boost [Xgboost] algorithm: 17

2.2 Performance metrics for binary classification: 20

2.2.1 Accuracy 20

2.2.2 Imbalance metrics 20

2.2.3 Probabilistic classifiers metrics 23

2.3 Model Evaluation 26

2.3.1 Hold−Out process 26

2.3.2 K-folds Cross−Validation 27

3 Proposed Experiment **30**

3.1 Datasets 30

3.2 Hyper-parameter Ranges 33

3.3 Meta-features Extraction 34

3.4 Genetic Algorithms 37

4 Results **44**

5 Conclusion **49**

5.1 Future work 49

Bibliography **51**

List of Figures

2.1 K−Nearest Neighbor algorithm[Ras17]. 1
2.2 K−NN accuracies with different K values 4
2.3 Logistic regression algorithm 5
2.4 L1 and L2 penalty with different values of C in Logistic regression[PVG+11] 8
2.5 Support Vector machine[Say11] 9
2.6 Non-Linear transformation in Support Vector machine[Say11] 10
2.7 Decision Tree Classification[Ras17]. 12
2.8 Ensemble learning Example on ozone and temperature datasets [Est17] . . . 15
2.9 Random Forest steps [MA15] 16
2.10 Speed Benchmark Result [CH15] 18
2.11 An example of ROC curve of the LogisticRegression classifier over 3-fold cross validation[Ras17] 24
2.12 precision-recall curve[BHK15] 25

3.1 Genetic Algorithm. Tree Basic steps of GA: selection, crossover and mutation. [HLL13]. 38
3.2 Single point crossover[SS07]. 41
3.3 Two-point crossover[SS07]. 42
3.4 Two-point crossover[SS07]. 42
3.5 Bit-flipping mutation of parent a to form offspring b[SS07]. 43

List of Tables

2.1 K-NN Implementation in Python Scikit-learn package 3
2.2 Logistic regression Implementation in Python Scikit-learn package 6
2.3 Comparison of L1 and L2 regularizations. 7
2.4 Support vector machine Implementation in Python Scikit-learn package . . 11
2.5 Decision Tree implementation in Python Scikit-learn package 14
2.6 Random Forest Implementation in Python Scikit-learn package 17
2.8 Hyper−parameters 22
2.9 AUC-ROC Score Implementation in Python Scikit-learn package 25
2.10 Average Precision Score Implementation in Python Scikit-learn package . . 26

3.1 The training datasets of the thesis 32
3.2 Hyper-parameters Ranges of the applied machine learning algorithms. . . . 34
3.3 Caption for LOF 35
3.4 MF used in the experiment 37

4.1 The MFs which are correlated with the accuracy performance metric. 45
4.2 The MFs which are correlated with the AUC-ROC metric. 45
4.3 The MFs which are correlated with the Cohen's kappa metric. 46
4.4 The MFs which are correlated with the F-measure metric. 46
4.5 The meta-features which are correlated with the Matthews correlation coefficient metric. 46
4.6 The MFs which are correlated with the precision performance metric. 47
4.7 The MFs which are correlated with the recall performance metric. 47
4.8 The MFs which are correlated with the balance accuracy metric 47

4.9 The common MFs that are correlated with different performance metrics . . 48

Listings

2.1 Different performance metrics used in this thesis from Sklearn package . . . 22
2.2 K-folds cross validation implementation in Python 27
2.3 K-folds function which divides the dataset to k-disjoint partitions. 28

Acronyms

AutoML **A**utomated **M**achine **L**earning.

GA **G**enetic **A**lgorithm.

HP **H**yper**P**arameter.

MF **M**eta **F**eature.

ML **M**achine **L**earning.

MtL **M**eta **L**earning.

Chapter 1

Introduction

1.1 Motivation

The most important task of data science and automated machine learning (AutoML)[1] is to find the best Machine learning (ML) technique to use for a given dataset in hand and tune its hyper-parameters (HPs). While HP tuning can largely affect the predictive performance of a used ML algorithm, this task is data-dependent such that a certain HP configuration might be very good in case of one dataset while not to other datasets.

The recommendation of the best ML algorithm and it's adequate HP configuration are the subject of a research area known as meta-learning (MtL), described in more details in [BPR09]. The first step in the use of MtL is the representation of each dataset by a vector of features, often named meta-features (MFs), which describe important aspects of a dataset and are used as predictive attributes in a meta-dataset. The next step is the recording of the predictive performance of a set of ML algorithms and their HP configurations on these datasets. The predictive performances for each dataset will be the target attribute for the MFs extracted from the dataset in the meta-dataset. Finally, a meta-model induced by the application of a classification or regression technique to the meta-dataset can be used to predict the most adequate target attribute (a ML technique and/or HP configuration) for a new dataset, using the MFs as predictive attributes.

Recent studies in the MtL literature have been concerned with investigating different types of MF extracted in order to characterize a dataset. These types, categorizing 88 or

[1] `http://www.ml4aad.org/automl/`

more different MFs are i) simple measures, such as number of classes, ii) statistics measures, such as skewness and kurtosis, iii) information theoretic measures, such as attribute entropy, iv) landmarking measures such as the performance and execution times of some basic ML algorithms on the dataset, v) features extracted from decision tree models, vi) measures that analyze the complexity of a problem and vii) measures based on complex network properties.

The experiments which are provided by the supervisor of this thesis that is not yet published showed that the choice of a suitable combination of MF types for HP recommendation seems to be data dependent and there is no MF type which is clearly superior to other MF types on all datasets. However, only the combinations of the 8 MF types were investigated, i.e. either none or all of the MFs belonging to the same MF type were included in the choice, leading to $2^8 - 1 = 255$ different combinations. To investigate the full range of various combinations of MFs, not only the combinations of MF types, would require more than 2^{80} cases to try what is infeasible. The goals of this thesis are to:

- propose an approach for investigating the utility of various MF combinations with means of genetic algorithms (GA);
- implement the proposed approach in the experiment involving different classification algorithms on various datasets used in that experiments;
- analyze the results of the experiment and provide insights for future research in MtL.

1.2 Thesis outline

The thesis is organized as follows:

- Chapter 2 provides an explanation to the machine learning algorithms that are used in this experiment.
- Chapter 3 gives an detailed explanation of the datasets that are used in this experiment, the hyper-parameters ranges for the machine learning algorithms, the meta-features that are extracted and used in this experiment and the genetic algorithm.
- Chapter 4 provides a detailed analysis for the results of this experiment.

- Chapter 5 draws our conclusions and provides the future work for this thesis.

Chapter 2

Machine learning

2.1 Binary classification Models

2.1.1 K-Nearest Neighbors

K-Nearest Neighbor is a supervised machine learning algorithm that can be used for classification and regression problems. It compares the new samples of the dataset to the existing ones that were kept during the prediction process. In fact, it doesn't make actual learning. That's why it is called a memory-based algorithm. Prediction in K-NN algorithm is easy in the way that for any new sample it looks for K most similar samples based on some distant metrics. After that, it assigns a class label to the new sample by a majority voting as it shown in the following figure.

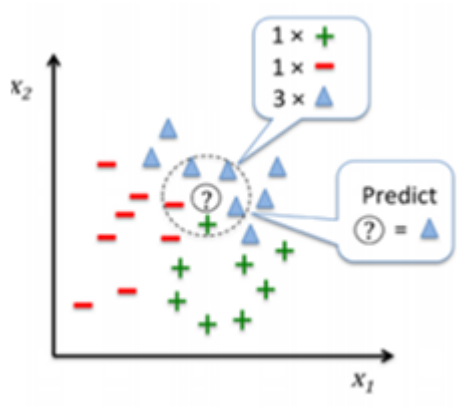

Figure 2.1: K−Nearest Neighbor algorithm[Ras17].

The triangle class label as shown in the above figure was assigned to the new data

sample(?) according to majority nearest neighbors. There are different distant functions to compare the similarity between the data samples which are the following:

1. Euclidean distance:

   Euclidean distance is the square root of the sum of the squared differences between two points **x,y** and it is calculated as follows:

$$D(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

2. Hamming Distance:

   Hamming Distance is the number of bits where two binary vectors differ and it is calculated as follows:

$$D(x,y) = \sum_{i=1}^{n}(x_i \oplus y_i)$$

   where $\oplus$ denotes the exclusive-or (XOR) operator.

3. Manhattan Distance:

   Manhattan Distance is the sum of the absolute difference between two points and it is calculated as follows:

$$D(x,y) = \sum_{i=1}^{n}|(x_i - y_i)|$$

4. Minkowski Distance:

   Minkowski Distance is the generalized metric distance of both Euclidean distance and Manhattan distance and is calculated as follows:

$$D(x,y) = (\sum_{i=1}^{n-1}|(x_i - y_i)|^p)^(1/p)$$

   - When p=1, it is equal to Manhattan distance.
   - When p=2, it is equal to Euclidean distance.

The Classification result depends on the selected distance function. In our implementation, we used KNeighborsClassifier that is available in scikit−learn package as it is shown in the following table:

KNeighborsClassifier[1]
python
A. Building the classifier: class KNeighborsClassifier(n_neighbors=5, weights='uniform', metric='minkowski') B. Parameters: − n_neighbors: number of neighbors. − weights: weight function used in prediction. C. Making prediction: Predict(X): Predict the class labels for the provided data. temp_y_test_pred = test_classifier.predict(validation_X_test) predict_proba(X): Return probability estimates for the test data X. y_score = test_classifier.predict_proba(validation_X_test)[:, 1]

Table 2.1: K-NN Implementation in Python Scikit-learn package

As it is shown in table 2.1, You can change the distance calculation for k−neighbors algorithm[KNeighborsClassifier] by using the weights parameter. The default value for it is uniform which weighs all data points equally. One can define his own function and pass it as a value for weights parameter.

K-Nearest Neighbors algorithm depends on the size of the neighborhood whose best value can be found by trying different values. In this research, we made a grid for this parameter and we tried with all values in that grid through k-fold cross validation process which is described in section 2.3.2. As the value of K increases, more neighbors that give smoother boundaries will be gotten. If it's too small, then the resulting model will be susceptible to the noise in the dataset. If K is set properly, a good generalization can be

[1]**More information can be found on the following website:
`http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html`

achieved.

It is shown in Figure 2.2 that K-NN classifier which is applied to backache dataset peaks in accuracy around K=5. After that, it becomes stable. X-axis in figure 2.2 refers to the number of neighbors and Y-axis refers to accuracy which is defined in 2.2.1.

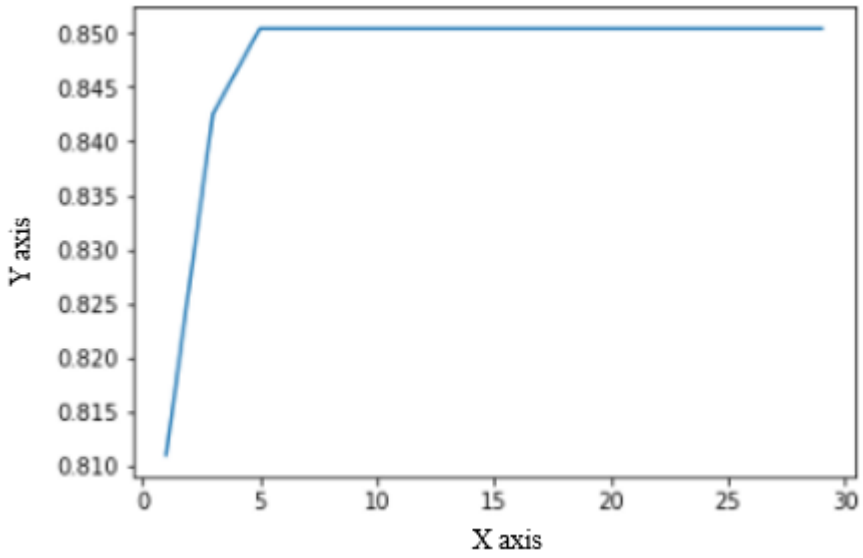

Figure 2.2: K−NN accuracies with different K values

In our experiment, we used a wide range of k values with K-fold cross validation process which is mentioned in section 3.2.

2.1.2 Logistic regression

Logistic regression is a binary classification algorithm that is used in different fields mainly in biostatistics. It is also used as a prototype for some classification algorithms. Logistic regression depends on a probability function that gives the input a chance to belong to any of the two classes. In this experiment, we deal with a binary classification problem [‘0’ or ‘1’].

The probability function in the logistic regression is the Sigmoid function which is:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta_0 + \theta x}}$$

where θ_0 is called bias. The result for the logistic regression function is always between zero and one as it is shown in the following figure for the range [-6,6]:

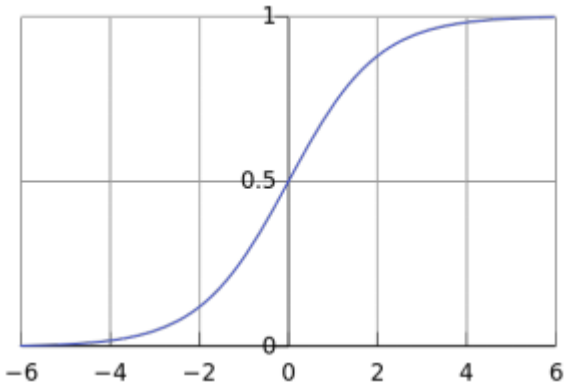

Figure 2.3: Logistic regression algorithm

The inverse of logistic regression function is the logit function which is the logarithm of the odds ratio[2]:

$$F(x) = ln\frac{F(x)}{1 - F(x)} = \theta_0 + \theta x$$

We use the scikit-learn implementation of logistic regression through LogisticRegression Class as shown in the following table:

LogisticRegression[3]
python
A. Building the classifier: class LogisticRegression(penalty='l2', C=1.0) B. Parameters: – penalty: penalization term. – C: Inverse of regularization strength C. Making prediction: Predict(X): Predict the class labels for the provided data. temp_y_test_pred = test_classifier.predict(validation_X_test) predict_proba(X): Return probability estimates for the test data X.

[2]The odds of an event is the probability of that event divided by the probability of its complement. It can found from the logit function by exponentiating that function.

[3]More information can be found on the following website:
`http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

y_score test_classifier.predict_proba(validation_X_test)[:, 1]

Table 2.2: Logistic regression Implementation in Python Scikit-learn package

Prediction rule:

It was mentioned that the output of the logistic regression model is a probability as it estimates the conditional distribution P(y|x). The logistic regression model is derived from the assumption that the labels are distributed according to Bernoulli distribution. Thus, the conditional probability of the positive class is d=P(1|x) where x is the feature vector and the conditional probability of the negative class is P(0|x)= 1-d. The linear combination $\theta_0 + \theta x$ identifies a hyperplane that separates the positive class from the negative one, so for any positive sample, we have $\theta_0 + \theta x \geq 0$ where we have $\theta_0 + \theta x \leq 0$ for the negative class.

Training algorithm:

Logistic regression minimizes its associated lost function[4] to provide the best hypothesis as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y^i \log(h_\theta(x^i)) + (1 - y^i) \log(1 - h_\theta(x^i))$$

where $h_\theta(x)$ is defined in equation 2.1.2 and y is the actual label which is zero in case of the negative class and one for the positive class.

Regularization:

In this experiment, L1 and L2 regularization techniques were used to avoid over-fitting problem in the Logistic regression models which can happen due to a large number of parameters and a small number of training samples. L1 is the sum of the weights while L2 is the sum of the squared weights. L2 regularized logistic regression solves the following optimization problem:

[4]loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. `https://en.wikipedia.org/wiki/Loss_function`

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

[PVG+11]

where w is the weights parameter, X_i denotes the value of the i^{th} attribute of X and C denotes the inverse of regularization strength. L1 regularized logistic regression solves the following problem:

$$\min_{w,c} ||w||_1 + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

[PVG+11]

The differences between L1 and L2 regularized logistic regressions are summarized in the following table:

L2 Regularization	L1 Regularization
Computationally efficient	Computationally inefficient
No sparse[5]outputs	Leads to sparse outputs

Table 2.3: Comparison of L1 and L2 regularizations.

As there is no rule to select the best values for C and penalty, we prepare a grid with a wide range of values and we try with all values in that grid through the K-folds cross validation process which is described in details in section 2.3.2. You can see in the following figure an example of classifying 8x8 images of digits into two classes: 0-4 against 5-9. The figure shows different values of C and penalty.

[5]Sparse means that most entries in a matrix (or vector) are zero and only very few entries is non-zero. L1-norm produces many coefficients with zero values.

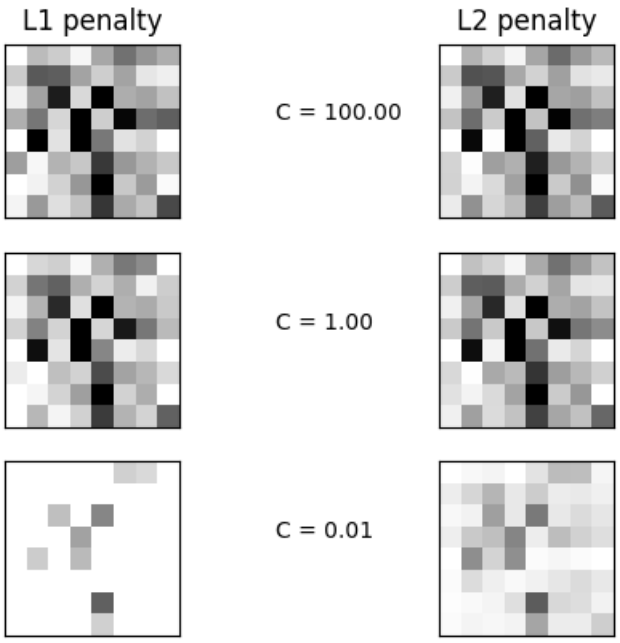

Figure 2.4: L1 and L2 penalty with different values of C in Logistic regression[PVG+11]

It is shown in the above figure that L1 penalty leads to sparse outputs while L2 doesn't.

2.1.3 Support vector machine:

Support vector machine is a supervised learning algorithm that can be used for classification and regression analysis. It works with the linear classification but can also perform a non-linear classification by a simple trick known as a kernel trick. In addition to that, it provides a good generalization error [the prediction error when applying it to unseen data] as it tries to find the best hyperplane that maximizes the margin between the two classes. Data points on the margin are known as support vectors as shown in the following figure:

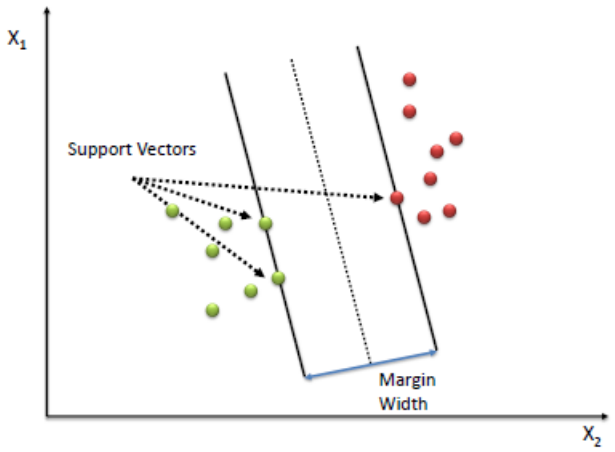

Figure 2.5: Support Vector machine[Say11]

The decision boundary [separating hyperplane] can be given by the following equation:

$$w \cdot x + b = 0$$

where w is the normal vector to the hyperplane and b is the bias[the offset of the hyperplane from the origin along the normal vector].

The support vectors define two planes parallel to the separating hyperplane: one hyperplane for the positive class described by the following equation:

$$w \cdot x + b \geq +1$$

and the other one for the negative class described by:

$$w \cdot x + b \leq -1$$

We can find w and b by solving the following optimization objective function:

$$min \frac{1}{2} \|w\|^2$$

$$subject\ to \qquad y_i(w \cdot x_i + b) \geq 1, \forall x_i \quad \text{[Say11]}$$

In case of non-linear data, the support vector machine handles it by using kernel func-

tion $K(x_i \cdot x_j)$ which transforms all data points to a higher dimensional feature space to make it linearly separable as shown in the figure 2.6. The kernel trick ensures that we don't need to transform these points to a higher dimensional space.

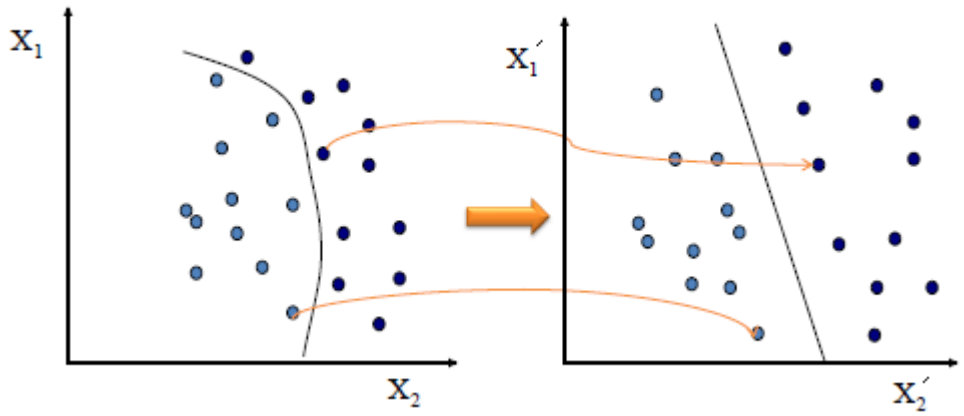

Figure 2.6: Non-Linear transformation in Support Vector machine[Say11]

The most used kernels are:

- Polynomial Kernel and is given by the following equation:

$$k(x_i \cdot x_j) = (x_i \cdot x_j)^d$$

  where d is the order of the polynomial.

- Gaussian or Radial Basis Function (RBF) and is given by the following equation:

$$\exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})\text{[Say11]}$$

  where σ is the width of the kernel.

In this experiment, we use different kernels that are specified by the keyword kernel as you can see in the following table

sklearn.svm.SVC[6]

[6]More information can be found on the following website:
`http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

python
A. Building the classifier: $-Linear : svm.SVC(kernel =' linear', C = 1.0)$ $-Polynomial : svm.SVC(kernel =' poly', degree = 3, C = 1.0, gamma =' auto')$ $-Rbf : svm.SVC(kernel =' rbf', gamma =' auto', C = 1.0)$ $-Sigmoid : svm.SVC(kernel =' sigmoid', gamma =' auto', C = 1.0)$ B. Parameters: – C: Penalty parameter C of the error term. – kernel : Specifies the kernel type to be used in the algorithm. – degree: Degree of the polynomial kernel function – gamma: Kernel coefficient for 'rbf', 'poly' and 'sigmoid' C. Making prediction: Predict(X): Predict the class labels for the provided data. temp_y_test_pred = test_classifier.predict(validation_X_test) decision_function(X): - Distance of the samples X to the separating hyperplane. y_score = test_classifier.decision_function(validation_X_test)[:, 1] Note: decision_function method is used to compute the ROC AUC score.

Table 2.4: Support vector machine Implementation in Python Scikit-learn package

2.1.4 Decision Trees

Decision tree algorithm is a supervised learning algorithm which is used to build classification and regression models in the form of a tree structure. The idea of this algorithm is to partition the training dataset into smaller subsets by splitting the features based on some threshold until you reach the leaf nodes. The resulting tree has decision nodes and leaf nodes. The Leaf node represents the class label while the decision node consists of two or more branches and each node in the branch corresponds to one feature.

This algorithm tries to put the best predictor of the training dataset at the root node of the tree in each level and after that the algorithm splits the training dataset to smaller subsets in a way that each subset will contain instances with similar values. This process will continue until we reach the leaf nodes for each branch of the tree as you can see in the

figure.

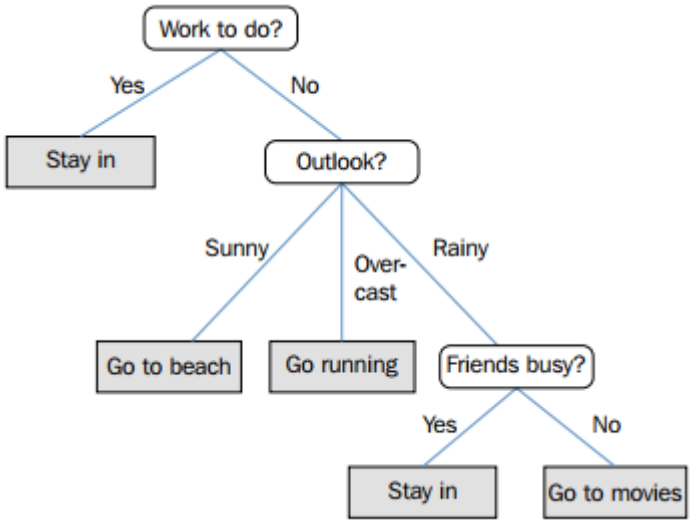

Figure 2.7: Decision Tree Classification[Ras17].

The most important thing in the decision tree implementation is to identify the best predictor attribute to be considered as a root node for each level of the tree structure [feature selection]. There are some measures to cope with the selection process which are:

- Entropy: Entropy for given node t is:

$$I_h(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \quad \text{[Ras17]}$$

  Note: p(i|t) is the relative frequency of class i at node t [MSK05].
  For binary classification problems, If only one class exists in the node[all of examples are positive or all of them are negative],then the entropy will be zero. If the number of the positive examples is equal to the number of the negatives then the entropy will be 0.5. For each possible split, the split that has the lowest entropy will be selected.

- Gini index:
  Gini index is a metric to check how often the random chosen element may be identified incorrectly. Thus, a feature that has a low Gini index will be selected.

Gini Index for a given node t is:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(-p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2 \quad \text{[Ras17]}$$

Note: p(i|t) is the relative frequency of class i at node t [MSK05].

Gini index will be 0.25 if the binary classes are balanced in a perfect way and it will be zero if all examples are either positive or negative.

Over-fitting in Decision Tree:

Over-fitting problem can happen in Decision Tree in case if the algorithm tries to go deeper in the partition process to reduce the training error but at the same time with an increased test error. According to that, It may lose the generalization capability. This may happen due to outliers and irregularities in the dataset.

There are two ways to avoid the over-fitting problem in Decision Tree:

- Pre-Pruning[top-down approach]:
  In this strategy, The algorithm stops the partition process of the dataset early if its goodness measure is less than the threshold value.

- Post-Pruning[bottom-up approach]:
  This strategy works in the opposite direction. It generates the complete tree and after that starts the elimination process of the nodes that their depth values[to the root node] is more than the threshold value.

In our implementation, we use DecisionTreeClassifier that is available in scikit-learn package as you can see in the following table:

DecisionTreeClassifier[7]
python
A. Building the classifier: $class\ \ DecisionTreeClassifier(criterion =' gini', max\_depth = None,$

[7]More information can be found on the following website:
`http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`

$min\_samples\_split = 2, min\_samples\_leaf = 1, max_features = None$)

B. Parameters:

– criterion: The function to measure the quality of a split.

– 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:

– max_features: .

The number of features to consider when looking for the best split

C. Making prediction:

Predict(X): Predict the class labels for the provided data.

temp_y_test_pred = test_classifier.predict(validation_X_test)

predict_proba(X): Return probability estimates for the test data X.

y_score = test_classifier.predict_proba(validation_X_test)[:, 1]

Note: predict_proba method is used to compute the AUC-ROC score.

Table 2.5: Decision Tree implementation in Python Scikit-learn package

2.1.5 Random Forest:

Random Forest is an ensemble learning algorithm that can be used for classification and regression problems. Ensemble learning means combining more than one model together to improve the generalization error. The Random Forest algorithm works with either continuous or categorical predictors. It operates by building multiple decision trees from the training dataset and the output is selected by the majority voting which means the most frequently predicted class.

The idea of the ensemble method is to combine the predictions from different machine learning algorithms instead of just use one algorithm to build a robust model. Each classifier alone is a weak leaner but when you combine a group of classifiers, you will get a strong learner as shown in the figure 2.8.

Each classifier in the figure referred by a gray curve is a weak learner while the classifier

referred by a red curve is a much better approximation to the data more than the other classifiers individually.

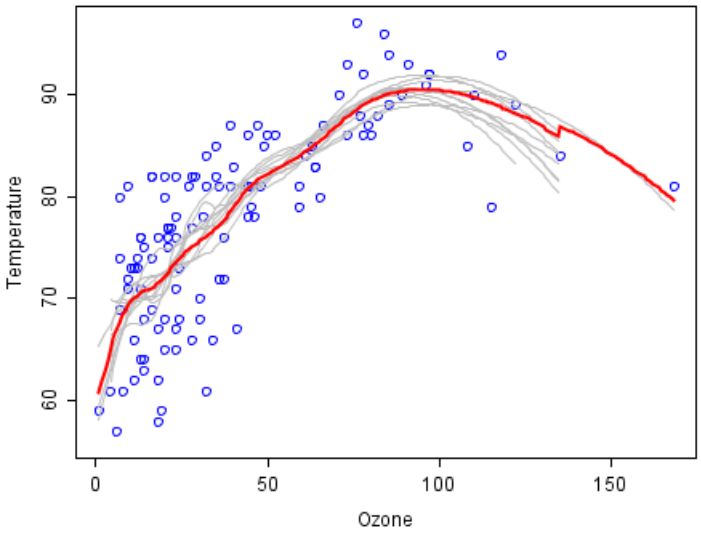

Figure 2.8: Ensemble learning Example on ozone and temperature datasets [Est17]

Random Forest works based on the bagging technique. Bagging technique works on that dataset as follows:

- Create bootstrap samples (random samples with replacement) from the training dataset.
- Decision tree Classifier is fitted on each bootstrap sample.
- the final prediction for the new samples can be taken according to the majority vote in the forest as shown in the figure 2.9

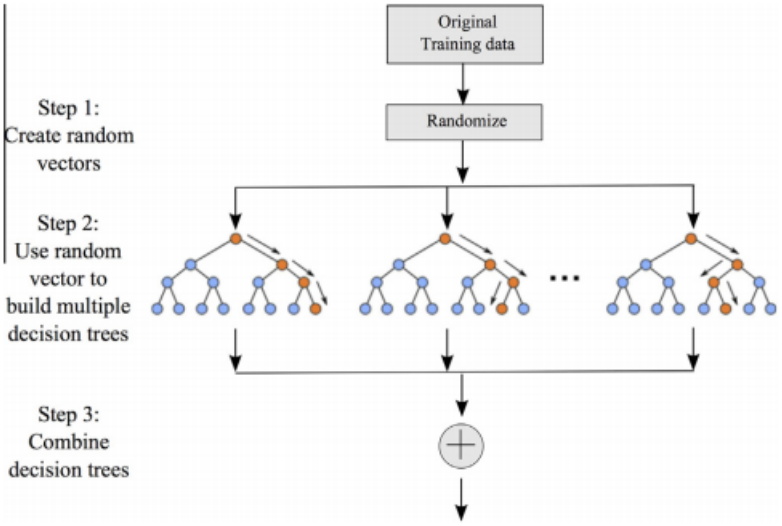

Figure 2.9: Random Forest steps [MA15]

Random Forest provides an improvement of the bagging technique by adding an additional randomization step for the features selection. The additional step tries to solve the problem of high correlations between the outputs of the different trees in the forest. If one or more features have a strong effect on the target output, then they will be selected in many trees which leads to a high correlation between them. As a result of that, we will get a model with poor predictive performance.

Instead of looking for all features in the dataset to select the best one for the splitting process as the decision tree did, the random forest adds a randomization step that each decision tree in the forest uses a random subset of features for selecting the optimal split point. As a result of that, the decision tree algorithm avoids choosing the strong features [predictors] many times, and thus we will get a model with an improved predictive performance.

The number of features for the Random forest algorithm is a hyper-parameter which means it should be chosen carefully as it affects the performance of the resulting model. In our implementation, we tried with different values for this hyper-parameter and for others as well. The details for them can be found in table 2.6.

RandomForestClassifier[8]

[8]More information can be found on the following website:

python
A. Building the classifier: $class\ \ RandomForestClassifier(n\_estimators = 10, criterion =' gini',$ $max\_depth = None, min\_samples\_split = 2, min\_samples\_leaf = 1,$ $max\_features =' auto', bootstrap = True)$ B. Parameters: – n_estimators :The number of trees in the forest. – Criterion: The function to measure the quality of a split – max_features: The number of features to consider when looking for the best split – 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. – bootstrap: Whether bootstrap samples are used when building trees. C. Making prediction: Predict(X): Predict the class labels for the provided data. temp_y_test_pred = test_classifier.predict(validation_X_test) predict_proba(X): Return probability estimates for the test data X. y_score = test_classifier.predict_proba(validation_X_test)[:, 1]

Table 2.6: Random Forest Implementation in Python Scikit-learn package

2.1.6 Extreme Gradient Boost [Xgboost] algorithm:

Xgboost is a software library that is an optimized version of Gradient Boosting algorithm. Gradient boosting machine is a supervised machine learning algorithm that can be used for classification and regression problems. It builds a prediction model by merging a

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

series of weak prediction models in an iterative way to build a stronger one.

Xgboost is available for Java, R, C++, Julia and even from command line interface(CLI). It's a more efficient implementation compared to gradient boosting algorithms. It is faster than other gradient boosting algorithms as shown in Figure 2.10.

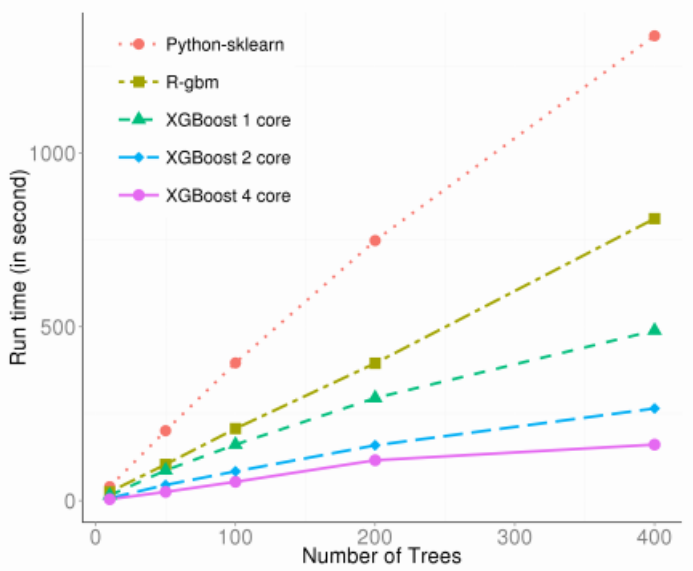

Figure 2.10: Speed Benchmark Result [CH15]

Xgboost is used for supervised learning problems which training data is used to predict a target value. It contains two kinds of solvers: the linear solver and the tree learning algorithms which makes it more efficient and faster compared to gradient boosting algorithms. It is used for a lot of competitions on kaggle and has a lot of advantages like:

1. Distributed processing: Xgboost can do a parallel computation on a single machine.

2. Dealing with missing values: Xgboost deals with missing values not like the other algorithms in which you must handle the missing values before applying the machine learning algorithms. In this experiment, the datasets doesn't have missing values.

3. Accuracy: Xgboost gives good results.

4. Regularization: Xgboost has a regularization parameter which is not found on Gradient Boosting implementation and because of that, we can overcome with the overfitting problem.

5. Xgboost has its own implementation of cross validation.

6. Feasibility: In Xgboost, you can use your own optimization objective function and evaluation.

In this experiment, we use the XGBClassifier class that is available in the xgboost library for python as shown in the table 2.7:

XGBClassifier[9]
python
A. Building the classifier: $class\ \ xgboost.XGBClassifier(max\_depth = 3, learning\_rate = 0.1,$ $n\_estimators = 100, gamma = 0, min\_child\_weight = 1,$ $subsample = 1, colsample\_bytree = 1, reg\_alpha = 0)$ B. Parameters: – max_depth: Maximum tree depth for base learners. – learning_rate: Boosting learning rate (xgb's "eta") – n_estimators: Number of boosted trees to fit. – objective: Specify the learning task and the corresponding learning objective or a custom objective function to be used. – gamma: Minimum loss reduction required to make a further partition on leaf node of the tree. – subsample: Subsample ratio of the training instance. – colsample_bytree: Subsample ratio of columns when constructing each tree. – reg_alpha: L1 regularization term on weights – reg_lambda: L2 regularization term on weights C. Making prediction: Predict(X): Predict the class labels for the provided data. temp_y_test_pred = test_classifier.predict(validation_X_test) predict_proba(X): Return probability estimates for the test data X. y_score = test_classifier.predict_proba(validation_X_test)[:, 1]

[9]More information can be found on the following website:
`http://xgboost.readthedocs.io/en/latest/python/python_api.html`

Note: predict_proba method is used to compute the AUC-ROC score.

Table 2.7: XGBoost classifier Implementation in Python

2.2 Performance metrics for binary classification:

Different metrics are used to measure the performance of a binary classifier.

2.2.1 Accuracy

Accuracy is the proportion of samples which are classified correctly among the whole samples of the dataset. It is calculated as follows:

$$F(X) = \frac{TN + TP}{TN + TP + FP + FN}$$

where:

- TP refers to True positive: As an example of it, if One has a positive result for a cancer disease test and he has that disease, then it's called a true positive.
- TN refers to True negative: As an example of it, if the result of a medical test for a cancer disease is negative for someone doesn't have that disease, then it's called a true negative.
- FP refers to False positive: As an example of it, if the result of a medical test for a cancer disease is positive for someone doesn't have that disease, then it's called a false negative.
- FN refers to False negative: As an example of it, if the result of a pregnancy test is negative and the woman is pregnant, then it's called a false negative.

2.2.2 Imbalance metrics

- Balance Accuracy:

  Balanced accuracy is the arithmetic mean of sensitivity and specificity as follows:

$$\frac{TP/(TP+FN)+TN/(TN+TP)}{2} = \frac{sensitivity + specificity}{2}$$

- Precision:

  Precision is a measure for the positive predictive value and is defined as follows:

$$Precision = \frac{TP}{TP+FP}$$

- Recall:

  Recall is a measure for the true positive rate and is defined as follows:

$$Recall = \frac{TP}{TP+FN}$$

  High precision means a low false positive rate and a high recall means a low false negative rate. High precision and High recall means that you have accurate results but if you have a high recall and low precision, then it means that most of the predicted values are false. At the same time, if you have a low recall and a high precision, then it means that most of the predicted values are correct. The best case for your model when you have a high precision and a high recall.One way to summarize both metrics: precision and recall is the F-score.

- F-score: F-score is a harmonic mean for both recall and precision as in the following:

$$F - score = \frac{2 * Precision * Recall}{Precision + Recall}$$

- Matthews Correlation Coefficient[MCC]:

  The Matthews correlation coefficient is a measure of the quality of binary classifications and it is counted as a balanced measure even though the sizes of the classes are different.It is calculated as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

  The value for MCC is between -1 and +1 in which -1 means an inverse prediction,

+1 means a perfect prediction and zero means an average random prediction.

- Cohen's kappa:

  Cohen's kappa is a statistic that measures inter-annotator agreement. Suppose you have two raters classify different objects into two categories Yes and No as it is shown in the following table:

		Rater 1	
		Yes	No
Rater2	Yes	TP	FN
	No	FP	TN

Table 2.8: Hyper−parameters

As it is shown in the table, Values on the main diagonal of the table represents the counts of agreements. Cohen's Kappa is calculated as follows:

$$\frac{n\sum_{i=1}^{C} x_{ii} - \sum_{i=1}^{C} x_i \cdot x_{\cdot}i}{n^2 - \sum_{i=1}^{C} x_i \cdot x_{\cdot}i} \qquad \text{[SGH15]}$$

where:

  - x_{ii} is the cell count in the main diagonal in Table 2.9.
  - n is the number of examples in the dataset.
  - C is the number of class labels.
  - $x_i, x_{\cdot}i$ are the rows and columns total counts respectively.

In this experiment, we use the sklearn package to import all metrics as shown in the following listing 2.1:

```
Average_precision_recall = average_precision_score(validation_y_test.ravel(),
                                                   y_score)
accuracy=metrics.accuracy_score(validation_y_test.ravel(), temp_y_test_pred)
Classification_auc = roc_auc_score(validation_y_test.ravel(), y_score)
F_measure=metrics.f1_score(validation_y_test.ravel(), temp_y_test_pred)
matthews_corr=matthews_corrcoef(validation_y_test.ravel(), temp_y_test_pred)
```

```
coh_kapp=cohen_kappa_score(validation_y_test.ravel(),temp_y_test_pred)
balance_accuracy=recall_score(validation_y_test.ravel(),
                              temp_y_test_pred,average='macro')
precision=metrics.precision_score(validation_y_test.ravel(), temp_y_test_pred)
call=metrics.recall_score(validation_y_test.ravel(), temp_y_test_pred)
```

Listing 2.1: Different performance metrics used in this thesis from Sklearn package

2.2.3 Probabilistic classifiers metrics

In case the output of the classifier is a probability, the performance metric can be: Receiver Operating Characteristics curve(ROC) or Precision-Recall curve which are described in this section.Most of classifiers provide either predict_prob or decision_function methods for evaluating the degree of certainty of your predictions.

In this experiment, we use predict_prob for knn, logistic regression, decision tree, random forest and neural network algorithm. For support vector machine, we use decision_function for calculating the area under curve and the average precision value.

- Receiver Operating Characteristics curve(ROC)

  A Receiver Operating Characteristics curve(ROC) is a graphical representation of the true positive rate against the false positive rate for different thresholds. ROC curve is used for visualizing and selecting the classifiers based on their performance.

  In Roc curve, x-axis represents the False positive rate and Y-axis represents the True positive rate as shown in the following Figure.

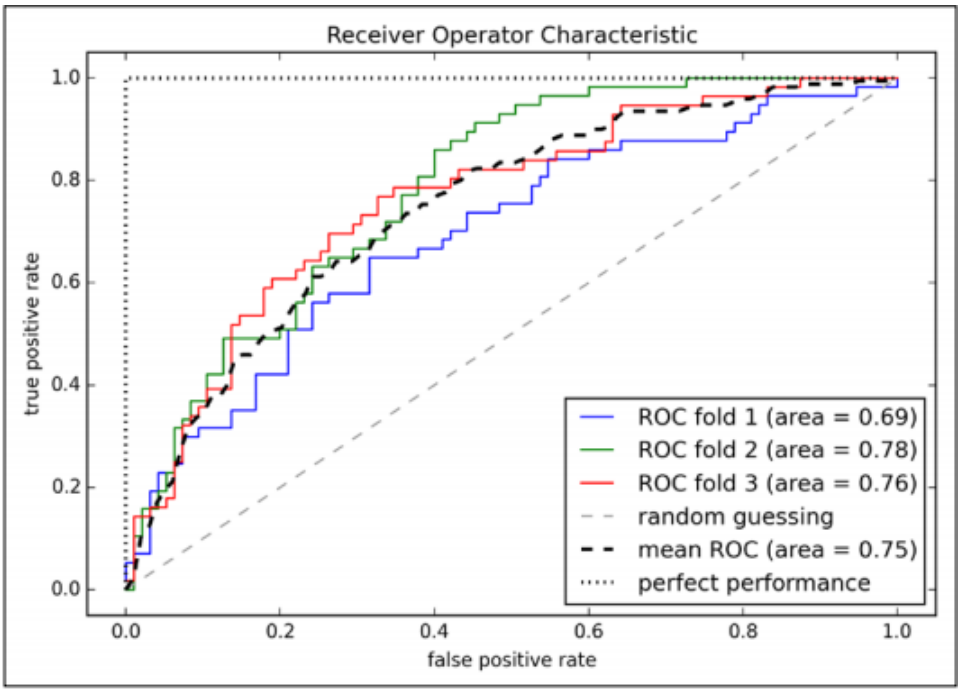

Figure 2.11: An example of ROC curve of the LogisticRegression classifier over 3-fold cross validation[Ras17]

As shown in figure 2.11 that for a perfect classification, which means no errors, the graph should be in the top-left corner in a position that True positive rate (TPR) equals one and False positive rate (FPR) equals zero.The diagonal dashed line is the ROC curve of a random classifier.

We can express the ROC curves numerically by calculating the area under it. It is usually referred by AUC-ROC. In this case, AUC score for a random classifier is equals to 0.5 and 1.0 for a perfect classifier.

In this experiment, python scikit-learn package is used for computing AUC score as it is shown in the following table:

AUC-ROC score[10]
python
A. Implementation: $sklearn.metrics.roc\_auc\_score(y\_true, y\_score, average =' macro',$

[10]More information can be found on the following website:
`http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html`

$sample\_weight = None$)

B. Parameters:

- y_true:True binary labels in binary label indicators.
- y_score: Target scores, In this experiment it is the probability estimates of the positive class and it is calculated as follows:

$y\_score = test\_classifier.predict\_proba(validation\_X\_test)[:, 1]$

Table 2.9: AUC-ROC Score Implementation in Python Scikit-learn package

- Precision-Recall curve:

  It's similar to the ROC curve, it is a graph of precision versus recall for different thresholds as it is shown in figure:

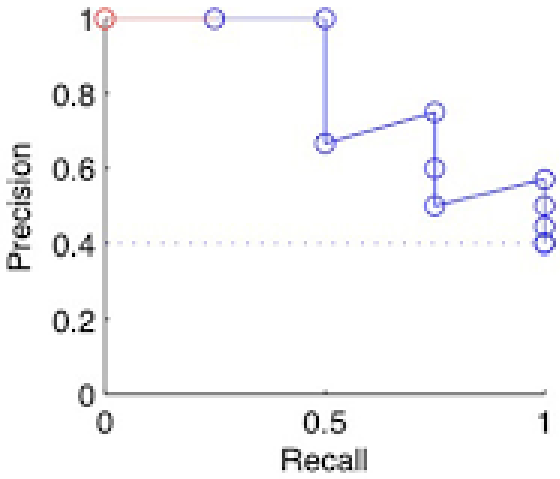

Figure 2.12: precision-recall curve[BHK15]

As it is shown in the Figure 2.12, the prefect classifier is located in the top-right corner of the graph. The dotted line of Precision-recall graph is a random classifier which has an average precision equal to the actual positive (P) divided by the sum of the actual positive (P) and the actual negative (N). As the area under ROC curve,The graph of the Precision-Recall can be expressed numerically by average precision score. For a perfect classifier it's equal 1.0 and 0.5 for a random classifier.

Average Precision score[11]

[11]More information can be found on the following website:

python
A. Implementation: $sklearn.metrics.average\_precision\_score(y\_true, y\_score, average =' macro',$ $sample\_weight = None)$ B. Parameters: – y_true:True binary labels in binary label indicators. – y_score: Target scores, In this experiment it is the probability estimates of the positive class and it is calculated as follows: $y\_score = test\_classifier.predict\_proba(validation\_X\_test)[:, 1]$

Table 2.10: Average Precision Score Implementation in Python Scikit-learn package

2.3 Model Evaluation

Model evaluation is an important part of the development process to get the best model that is neither biased nor overoptimistic. In data-mining, you can't use the same data for the training and evaluation process because your model will be over-fitted or may be over optimistic. To avoid that issue, different methods use another dataset called test dataset [unseen data] for the evaluation. These methods are the following:

2.3.1 Hold−Out process

In this technique, the dataset is divided to different parts: one is the training set used to build the model and the other one is the validation set which is used for the evaluation process to ensure that the model is not over-fitted. The validation set is different form the training set so we are sure that we use unseen data for the evaluation process that will give us a less biased model. Over-fitting can be detected if the training performance increases while the validation performance decreases.

`http://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html`

2.3.2 K-folds Cross−Validation

In this way, the dataset is divided into k-disjoint partitions called folds which are roughly equal in size . One of the k-folds is used as a validation set while the remaining k-1 folds are combined to form the training set. This is done K times, in each time, one of the K-folds leaves out for the validation as follows [Alp10]:

$$V_1 = X_1 \qquad T_1 = X_2 \cup X_3 \cup ... \cup X_K$$
$$V_2 = X_2 \qquad T_2 = X_1 \cup X_3 \cup ... \cup X_K$$
$$.....$$
$$.....$$
$$.....$$
$$V_K = X_K \qquad T_K = X_1 \cup X_3 \cup ... \cup X_{K-1}$$

In this research's experiment, the K-folds cross validation is done as shown in the following listing 2.2:

```
for fold in range(num_folds):
   validation_X_test = X_train_folds[fold][12]
   validation_X_test=np[13].array(validation_X_test)
   validation_y_test = y_train_folds[fold]
   validation_y_test=np.array(validation_y_test)
   temp_X_train = np.concatenate(X_train_folds[:fold] +
                                 X_train_folds[fold + 1:])
   temp_y_train = np.concatenate(y_train_folds[:fold] +
                                 y_train_folds[fold + 1:])
```

Listing 2.2: K-folds cross validation implementation in Python

Imbalanced data refers to a problem with classification tasks where the classes are not represented equally. As the imbalance issue for the binary classes occurs in the datasets, we try to keep each fold to have different classes of positives and negatives to calculate the

[12] In each fold, we take one fold from the training dataset
[13] This is the NumPy package for scientific computing with Python. http://www.numpy.org/

area under the curve and to get a better model.

For dividing the dataset to different k-folds, I wrote my own implementation which divides the dataset to positives and negatives in which a good ratio of both classes has been achieved for each fold.

If you have for example 4 negatives and 2 positives, then the number of folds will be the smallest one which is 2 and the two positives will be distributed among the two folds, but if you have 10 positives and 6 negatives, in this case the number of folds will be 5 as we use the five folds for all datasets that have more than five negatives or five positives. You can see the implementation of the method that divides our datasets to k-folds in the following listing 2.3:

```
def k_folds(ndata,pdata):
  lp=pdata[14].shape[0]
  ln=ndata[15].shape[0]
  if lp<ln:
      if lp>=5:
          nfolds=5[16]
      else:
          nfolds=lp
  else:
      if ln>=5:
          nfolds=5
      else:
          nfolds=ln
  avg1 = ln / nfolds
  avg2 = lp / nfolds
  out1 = []
  out2=[]
  finalout2=[]
  last1 = 0
```

[14] refers to the positive part of the dataset in which the class label equals 1
[15] refers to the negative part of the dataset in which the class label equals 0
[16] max number of folds which is used in this experiment is 5

```
last2=0
i=0
while last1 < ln:
    out1.append(ndata[int(last1):int(last1 + avg1)])
    last1 += avg1

while last2 < lp:
    out2.append(pdata[int(last2):int(last2 + avg2)])
    last2 += avg2
for i in range(len(out1)):
    f=pd.concat([out1[i],out2[i]])
    finalout2.append(f)
return (finalout2,nfolds)
```

Listing 2.3: K-folds function which divides the dataset to k-disjoint partitions.

Chapter 3

Proposed Experiment

3.1 Datasets

The description of the 62 datasets from different sources [BL14] which are used for this experiment is shown in table 3.1. Each dataset is described with its name, the number of instances in it (Cases), the number of numerical attributed (Num), the number of categorical attributes (Cat), the number of the positive instances for the target attribute(Positive) and the number of the negative instances for the target attribute(Negative).

Name	Cases	Num	Cat	Positive	Negative
acuteinflammationsnephritis	99	1	5	44	55
analcatdata_boxing1	120	0	3	78	42
analcatdata_boxing2	132	0	3	61	71
analcatdata_creditscore	100	3	3	73	27
analcatdata_lawsuit	263	3	1	19	244
Appendicitis	106	7	0	21	85
autoUniv−au1−1000	997	20	0	739	258
Backache	180	5	26	25	155
Banana	5292	2	0	2373	2919
bank−marketing	4521	7	9	521	4000
banknote−authentication	1348	4	0	610	738

blood−transfusion−service	533	4	0	149	384
breast−cancer−wisconsin	463	9	0	238	225
Bupa	341	6	0	199	142
climate−simulation−craches	540	20	0	494	46
connectionist−mines−vs−rocks	208	6	0	97	111
fertility−diagnosis	100	9	0	12	88
habermans−survival	289	3	0	79	210
heart−disease−processed−hungarian	293	13	0	106	187
Hepatitis	155	6	13	123	32
hill−valley−with−noise	1212	100	0	606	606
horse−colic−surgical	300	13	14	109	191
indian−liver−patient	570	9	1	164	406
Ionosphere	350	33	0	225	125
kr−vs−kp	3196	0	36	1527	1669
leukemia−haslinger	100	50	0	51	49
lsvt−voice−rehabilitation	126	307	0	84	42
mammographic−mass	689	5	0	321	368
molecular−promotor−gene	106	0	57	53	53
Monks1	432	6	0	216	216
Monks2	432	0	6	142	290
Monks3	438	6	0	229	209
Mushroom	8124	0	21	3916	4208
Ozone−eighthr	2526	72	0	160	2366
ozone−onehr	2528	72	0	73	2455
Parkinsons	195	22	0	147	48
Phoneme	5395	5	0	1577	3818
pima−indians−diabetes	768	8	0	268	500
planning−relax	176	12	0	50	126

prnn_crabs	200	6	1	100	100
qsar−biodegradation	1052	41	0	354	698
qualitative−bankruptcy	103	0	6	78	25
Ringnorm	7400	20	0	3736	3664
Saheart	462	8	1	160	302
seismic−bumps	2578	11	4	2408	170
Spambase	4210	57	0	1679	2531
spectf−heart	267	44	0	212	55
spect−heart	228	0	22	101	127
statlog−australian−credit	690	8	6	307	383
statlog−german−credit	1000	7	13	300	700
statlog−german−credit−numeric	1000	24	0	300	700
statlog−heart	270	13	0	120	150
steel−plates−faults	1941	33	0	673	1268
thoracic−surgery	470	3	13	400	70
steel−plates−faults	1941	33	0	673	1268
thyroid−hypothyroid	3086	7	18	2945	141
thyroid−sick−euthyroid	3086	7	18	282	2803
tic−tac−toe	958	0	9	626	332
Voting	281	0	16	92	189
Wdbc	569	30	0	212	357
wholesale−channel	440	7	0	142	298
Wilt	4819	5	0	257	4562

Table 3.1: The training datasets of the thesis

3.2 Hyper-parameter Ranges

The distribution of all hyper-parameters that were used in this experiment are described in the following table 3.2. It were selected from different research papers which are cited at each algorithm.

Name	Parameters	Range
K-NN [BB10] [BS09] [ZX17]	k	[1,51]
Support Vector Machine [ISW16] [LMK11]	C Gamma Kernel Degree	$[10^{-3} - 10^{4}]$ $[10^{-3} - 10^{4}]$ Linear, sigmoid, rbf, poly 1-5
Random Forest [LKRK13] [FE17] [PBGS16] [EOY17] [DAG15]	n_estimators max_features min_samples_leaf max_depth min_samples_split bootstrap, criterion	[1,10000] [0.1,50] [1,20] [1,100,None] [2,64] [True, False] ["gini", "entropy"]
Xgboost [SW17] [RB17] [ZX17]	n_estimators learning_rate max_depth min_child_weight gamma reg_alpha colsample_bytree subsample	$[10, 1200]$ $[0.0001, 0.9]$ $[2, 15]$ $[1, 6]$ $[0.000001, 1]$ $[0.00001, 100]$ $[0.1, 1]$ $[0.1, 1]$

Decision Tree [ZX17] [DF13] [FE17]	max_features min_samples_leaf max_depth min_samples_split criterion	[0.1,16] [1,21] [1,51] [2,100] ["gini", "entropy"]
Logistic Regression [LMK11] [JZ16]	logistic_C logistic_penalty	$[10^{-3}, 10^{3}]$ ["$l1$","$l2$"]

Table 3.2: Hyper-parameters Ranges of the applied machine learning algorithms.

3.3 Meta-features Extraction

Meta-learning is an approach for solving the algorithm selection and recommendation problem. It is used for selecting a suitable machine learning model with best performance and/or Hyper-parameter(HP) configuration. Meta-features are measures describing dataset properties and characteristics. They are used in a meta-learning domain. The first step in Meta-learning is the representation of each dataset by a vector of Meta-features. The next step is to recode the predictive performance of different machine learning algorithms on these datasets and after that, a meta-model induced by the application of a classification or regression technique to the meta-dataset can be used to predict the most adequate target attribute which can be a machine learning technique and/or HP configuration for a new dataset.

Three requirements should be taken into account for the development of MFs which are:

1. Discriminative power: the MFs should have a good discriminative power according to the given MtL task.

2. Computational complexity: The extraction process of the MFs shouldn't be too complex. It should be maximum O(n log n) as pointed out by Pfahringer et al. [BPGC00].

3. Dimensionality: The number of MFs shouldn't be too large compared to the number of datasets used to avoid the over-fitting problem.

The main goal of this thesis is to find those MFs which are correlated well with the performance metrics of different machine learning algorithms that are used. The MFs are extracted using the mfe package[1] which is implemented in R.

Six types of the MFs are used in this experiment which are summarized in the tables 3.3 and 3.4.

MF type	Abbr	#	Description
General	GL		General information related to the dataset, also known as simple measures, such as number of instances, attributes and classes.
Statistical	ST		Standard statistical measures to describe the numerical properties of a distribution of data.
Discriminant	DT		Measures computed using the discriminant analysis.
Information-theoretic	IT		Particularly appropriate to describe discrete (categorical) attributes and their relationship with the classes.
Model-based	MB		Measures designed to extract characteristics like the depth, the shape and size of a Decision Tree model induced from a dataset.
Landmarking	LM		Represents the performance of some simple and efficient learning algorithms.

Table 3.3: MF types used in this experiment with their abbreviations[2]

Most studies in MtL focus on investigating different types of MFs for data characterization. For data characterization, three main types of MFs can be identified which are: simple, statistical and information-theoretic measures, land-markers and model-based measures. Simple, statistical and information-theoretical MFs are extensively used in MtL to summarize the dataset. However, new types of MFs have been proposed based on some approaches like model-based approach [HBK00] in which the model is built from the dataset

[1] Rivolli,A.,Garcia,L.P.F.,deCarvalho,A.C.P.L.F.(2017).mfe:Meta-FeatureExtractor. Rpackageversion0.1.0.http://CRAN.R-project.org/package=mfe

[2] https://cran.r-project.org/web/packages/mfe/vignettes/mfe-vignette.html

and the meta-features are the characteristics of that model. An example of this approach is the number of leaf nodes in a decision tree. In MtL, this approach is useful for algorithm recommendation.

Another apporach is the use of landmarking measures [FP01]. They evaluate the performance of different ML techniques on a given dataset. These measures are obtained by running simple and fast versions of the algorithms such as decision stumps [SW10].

Type	MF
GL	defective instances,dimensionality,% samples belongs to majority class,% missing values, total # attributes, total # binary attributes,total # classes, total # instances,# numeric attributes, # categorical attributes,% binary attributes,% numeric attributes, % symbolic attributes, std class distribution.
ST	absolute correlation, absolute covariance, degree of discreetness, geometric mean, harmonic mean, interquartile range divided by the standard deviation, kurtosis, median absolute deviation, normality, outliers, skewness, standard deviation, trim mean ,variance.
DT	cancor, cancor fract, center of gravity,discfct,eigen.fract, max eigenvalue,min eighenvalue,sdratio,wlambda
IT	attributes.concentration, attribute entropy, class.concentration, class entropy, equivalent.attributes, joint entropy, ,mutual information, noise to signal ratio
MB	average leaf corrobation, branch length, depth, homogeneity, max depth,# leaves,# nodes, nodes per attribute, nodes per instance, nodes per level, ,repeated nodes, shape, variable importance
LM	Decision Stump, Elite 1-NN performance, Linear Discriminant performance, Naive Bayes performance, 1-NN performance, worst node

Table 3.4: MF used in the experiment

3.4 Genetic Algorithms

Genetic Algorithm (GA) is a heuristic search-based optimization technique based on a natural selection process and evolutionary biology. It is used to find optimized solutions for difficult problems which can take a long time to be solved. Genetic algorithms are able to solve constrained and unconstrained optimization problems.

In this thesis, we used the genetic algorithm to find the MFs which are correlated well with the performance metrics of different machine learning algorithms.

An exhaustive selection of the MFs will result in a lot of different combinations. These combinations will be 2^n where n is the number of the MFs. In fact, we can't try all combinations which require a lot of computational work and that's why genetic algorithm is used for this kind of problems to select the best combinations.

In genetic algorithm, we have a population of possible solutions (individuals) for a given problem. For each generation, these individuals undergo recombination and mutation like in natural genetics to produce new solutions which are better suited to the problem. Each individual is assigned to a fitness value which is different according to the problem. A higher chance is given to fitter individuals to mate and produce more sophisticated solutions. The process of genetic algorithm is described in the following figure:

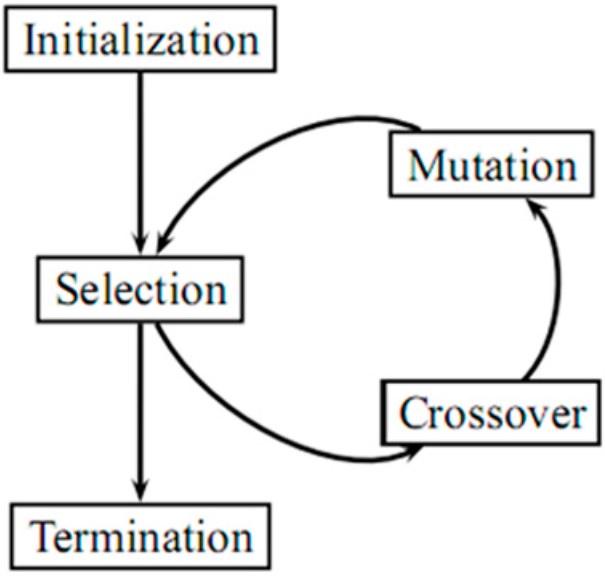

Figure 3.1: Genetic Algorithm. Tree Basic steps of GA: selection, crossover and mutation. [HLL13].

In this research, individuals are represented using a binary representation. The number of genes represent the number of MFs (length of the individual) which is in our case 88 MFs. The population size is set to be 100 and the number of generations is set to be 200.

1. Initialization:

   The first step in genetic algorithm is to create and initialize the individuals, referred to as chromosomes. In our experiment, we used DEAP [FDG+12] framework to implement the genetic algorithm. The initialization process for the individuals and the population is represented in the following code:

```
toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.attr_bool()
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_bool, 88)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

   As genetic algorithm is a stochastic method, the genes are set randomly. The individual in our problem looks like:

[1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1]

The individual is represented by 88 binary genes as we have 88 MFs. The positive value [1] means that the corresponding MF is selected whereas the negative value [0] means that the corresponding MF is not.

2. Selection:

   After the population and the individuals are created and initialized, we implement the fitness function. Fitness function is a function which takes an individual [candidate solution] as an input and produces the output which decides if this candidate fits with the problem or not. The implementation of the fitness function is in the following code:

```
def eval_func(individual):
  f1=[]
  f2=[]
  r_table=[]
  for i in range(len(dfm[3])):
      a1=algorithm_sim[4][algorithm_sim.columns.values[0]].
                        isin([dfm.iloc[i][0]])
      a4=algorithm_sim[algorithm_sim.columns.values[1]].
                        isin([dfm.iloc[i][0]])
      for j in range(len(dfm)):
          if i>=j:[5]
              a2=algorithm_sim[algorithm_sim.columns.values[1]].
                                isin([dfm.iloc[j][0]])
```

[3] dfm refers to the meta-dataset.

[4] algorithm_sim refers to the similarity file which contains all possible combinations between the datasets. The similarity is calculated according to the performance metric of different machine learning algorithms. The first and the second columns are the datasets names while the last column represents the similarity of the corresponding datasets in those two columns.

[5] This condition is to be sure that the current combination is unique and not duplicated. For example, the case i=2, j=1 which means that the current combination is D2,D1 will be taken, but the case i=1, j=2 which means that the combination is D1,D2 will be rejected because it's the same as D2,D1.

```
            a3=algorithm_sim[algorithm_sim.columns.values[0]].
                            isin([dfm.iloc[j][0]])

            for k1 in range(len(individual)):[6]
                if individual[k1]==1:
                    f1.append(dfm.iloc[i][k1+1])
                    f2.append(dfm.iloc[j][k1+1])

            if len(algorithm_sim[(a1)&(a2)])>=1:
                r_table.append([dfm.iloc[i][0],dfm.iloc[j][0],
                                cosine_similarity(f1,f2),
                               algorithm_sim[(a1)&(a2)].iloc[0]['sim_value']][7]
                              ])
            elif len(algorithm_sim[(a3)&(a4)])>=1:
                r_table.append([dfm.iloc[j][0],dfm.iloc[i][0],
                                cosine_similarity(f1,f2),
                               algorithm_sim[(a3)&(a4)].iloc[0]['sim_value']
                              ])
        del f2[:]
    del f1[:]
df1 = pd.DataFrame(data=r_table)
corr_value=df1.iloc[:,2].corr(df1.iloc[:,3])
return corr_value,
```

It is shown in the above code that the fitness value is the correlation between the following similarities:

- the similarities between the datasets based on the selected MFs which are set to one in the individuals.
- the similarities between the datasets based on the performance metric for each hyper-parameters combinations for the logistic regression model.

[6]The goal of this loop is to take the values of the MFs in which the genes are set to one.
[7]is the value of the similarity between the two datasets based on different performance metric

The individuals with greater fitness will have a greater probability to be selected for recombination. The selection operator chooses the individuals for the next generation based on their fitness level. The number of selected individuals is N/2 where N is the population size.

In this research, the selection is done based on a Tournament selection. In Tournament selection, a number Tour of individuals is selected randomly and the best one is chosen as a parent. The tournament size Tour is the parameter for the tournament selection and it takes a value from 2 to number of individuals in the population. If the tournament size is large, then the weak individuals will have a smaller chance to be selected. In our implementation, we use a tournsize equals to 4 as follows:

```
toolbox.register("select", tools.selTournament, tournsize=4)
```

3. Crossover:
   crossover is a genetic operator which takes more than one parent based on the selection operation and produces a new child from them. There are different kinds of crossover operators that can be used which are the following:

   - One Point Crossover:
     In one-point crossover, a random single crossover point on both parents organism is selected and after that all data beyond that point will be swapped between the two parent organisms get new off-springs as follows:

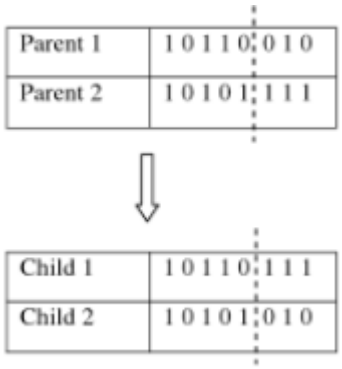

Figure 3.2: Single point crossover[SS07].

   - Two Point Crossover:
     In two-point crossover, two points are randomly selected on both parent organ-

isms and after that all the data between those two points are swapped to get new off-springs as follows:

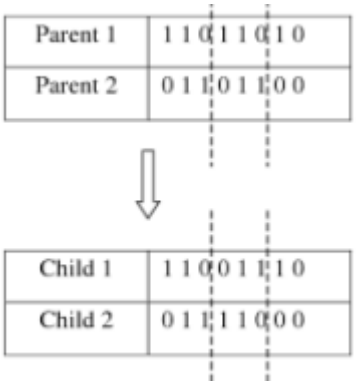

Figure 3.3: Two-point crossover[SS07].

In our implementation, we use two-point crossover as follows:
toolbox.register("mate", tools.cxTwoPoint)

- Uniform Crossover:
  In uniform crossover, for each pair of parents, a new crossover mask which has the same length as chromosomes is created randomly. The resulting offspring will be as follows:
  - if the value is one in the mask , then the gene is copied or swapped from the first parent
  - if the value is zero in the mask, then the gene is copied or swapped from the second parent.

  This means that the offsprings contain mixture genes from each parent as it is shown in the following figure:

Parent 1	1 0 1 1 0 0 1 1
Parent 2	0 0 0 1 1 0 1 0
Mask	1 1 0 1 0 1 1 0
Child 1	1 0 0 1 1 0 1 0
Child 2	0 0 1 1 0 0 1 1

Figure 3.4: Two-point crossover[SS07].

4. Mutation:

   The resulting off-springs in the crossover operation may be similar to the parents which will cause a new generation with low diversity. This issue is solved by a mutation operator in which some MFs in the off-springs will be changed randomly. The MF will be mutated if the generated random number which is between 0 and 1 is lower than the mutation rate which will be defined by the user, then the MF will be flipped as it is shown in the following figure:

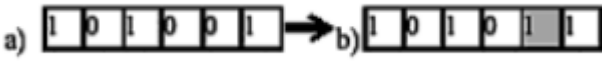

Figure 3.5: Bit-flipping mutation of parent a to form offspring b[SS07].

Chapter 4

Results

In This chapter, we present the results of our experiment. The experiment was divided into four steps as follows:

1. Applying different machine learning algorithms on our datasets which are mentioned in chapter 2.

2. Extracting the MFs which are mentioned in section 3.3.

3. Applying the genetic algorithm based on the results of steps 1 and 2 eight times as we have eight different performance metrics for each machine learning algorithm. It is mentioned in section 3.4.

4. Applying the frequent MFs mining algorithm on the results of step 3 eight times as we have eight different results from step 3. Frequent item set mining is one of the best known and most popular data mining techniques designed to identify elements that frequently co-occur.

The results of the machine learning algorithms are the measured values of different performance metrics which are mentioned in section 2.2 . These results and the MFs are an input for the genetic algorithm as mentioned in section 3.4. We run the genetic algorithms eight times as we have eight performance metrics which are: balance accuracy, Accuracy, F-measure, Cohen's kappa coefficient, Matthews correlation coefficient, recall, precision and average precision recall.

The last resulting population of the genetic algorithm is counted as the best one. As it is mentioned in section 3.4, we set the population size and the number of generations as 100. After that, The result of the genetic algorithm was the input for the frequent item set mining which in our case is the FP-Growth algorithm. The support rate for this algorithm was set to be 100. As we have eight different results from the genetic algorithm according to the different performance metrics , we run the frequent itemmset mining algorithm eight times.

The result of the frequent itemset mining for the genetic algorithm according to accuracy performance metric is as follows:

MF Name:
model.based.nodes.per.instance

Table 4.1: The MFs which are correlated with the accuracy performance metric.

As it is shown in table 4.1, there is just one MF which correlates with the accuracy performance metric. Of course, if we set the minimum support less than 100 , we will get more MFs.

For AUC-ROC metric, we got more MFs as it shown in the following table:

MF Name:
infotheo.attribute.entropy.sd
landmarking.decision.stumps.sd
landmarking.naive.bayes.sd
landmarking.worst.node.mean
model.based.nodes.per.level.sd
model.based.shape.mean
statistical.iqr.sd
statistical.trim.mean.mean

Table 4.2: The MFs which are correlated with the AUC-ROC metric.

As it is shown in the previous tables of 4.1 and 4.2 , there is no common MF for AUC-ROC and accuracy performance metrics.

The result of the Cohen's kappa metric is in the following table:

MF Name:
discriminant.cancor
infotheo.class.concentration.mean
infotheo.class.concentration.sd
landmarking.elite.nearest.neighbor.sd
model.based.shape.sd
statistical.correlation.sd
statistical.normality.sd
statistical.outliers.sd

Table 4.3: The MFs which are correlated with the Cohen's kappa metric.

The result according to F-measure performance metric is as follows:

MF Name:
statistical.outliers.mean

Table 4.4: The MFs which are correlated with the F-measure metric.

The resulting MFs of the genetic algorithm according to the Matthews correlation coefficient are as follows:

MF Name:
discriminant.min.eighenvalue
model.based.variable.importance.sd
statistical.iqr.sd

Table 4.5: The meta-features which are correlated with the Matthews correlation coefficient metric.

The resulting MFs of the genetic algorithm according to precision are as follows:

MF Name:

infotheo.attributes.concentration.mean
statistical.normality.sd

Table 4.6: The MFs which are correlated with the precision performance metric.

The resulting MFs of the genetic algorithm according with recall are as follows:

MF Name:
discriminant.cancor
infotheo.attributes.concentration.mean
infotheo.attributes.concentration.sd
infotheo.joint.entropy.mean
landmarking.elite.nearest.neighbor.sd
landmarking.linear.discriminant.sd
landmarking.worst.node.sd
statistical.outliers.sd

Table 4.7: The MFs which are correlated with the recall performance metric.

The resulting MFs of the genetic algorithm according to the balance accuracy coefficient are as follows:

Meta-feature Name:
discriminant.min.eighenvalue

Table 4.8: The MFs which are correlated with the balance accuracy metric

From the previous results, we found that some MFs are common in correlation with different performance metrics as follows:

MF Name:	Performance metric Name
discriminant.cancor	Cohen's kappa, recall
landmarking.elite.nearest.neighbor.sd	Cohen's kappa, recall
statistical.outliers.sd	Cohen's kappa, recall
infotheo.attributes.concentration.mean	recall, precision

statistical.iqr.sd	AUC-ROC, Matthews correlation coefficient
statistical.normality.sd	Cohen's kappa, precision
discriminant.min.eighenvalue	balance accuracy, Matthews correlation coefficient

Table 4.9: The common MFs that are correlated with different performance metrics

From the results, it is clear that only a few MFs are correlated with more than one performance metric. The only meta-features which are correlated with the same combination of the performance metrics [Cohen's kappa, recall] are:

- discriminant.cancor
- landmarking.elite.nearest.neighbor.sd
- statistical.outliers.sd

The other resulting MFs which are correlated with different combinations of performance metrics are:

- infotheo.attributes.concentration.mean
- statistical.iqr.sd
- statistical.normality.sd
- discriminant.min.eighenvalue

Chapter 5

Conclusion

The objective of this study is to investigate the importance of MFs for classification tasks in meta-learning. We conducted experiments using state of the art machine learning models. Furthermore, we tried to take care of the imbalance issue in our dataset. The datasets were preprocessed in a way that categorical attribute were converted to numerical attributes. Until the time of writing this thesis, this experiment has only been done for the logistic regression algorithm. The process for other machine learning algorithms is still running on the Linux server which is provided by ELTE university.

In this experiment, we extracted the metafeatures using mfe package in R and we neglected the ones that have the same value in which we are not interested. After that, we applied the genetic algorithm for finding the best combinations of the MFs because we can't try with all combinations as we have 88 MFs($2^8 = 256$). The last step of our work is to find the frequent MFs which are correlated with different performance metrics. We found that some MFs are common in correlation with some performance metrics.

5.1 Future work

Our work is only finished for the logistic regression algorithm and the other machine learning algorithms are still running on the server which means that this experiment is in progress. Thus, we are going to finish it and try to add the neural networks algorithm to this experiment. Furthermore, This project can be extended by solving the problem of the imbalance datasets which we have. In addition to that, we plan to run the genetic

algorithm with different values of different parameters and discover the patterns of the resulting MFs.

Bibliography

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning.* Massachusetts Institute of Technology, second edition edition, 2010.

[BB10] Christoph Bernau and Anne-Laure Boulesteix. Variable Selection and Parameter Tuning in High-Dimensional Prediction . *Technical Report Number 076, 2009*, 2010.

[BHK15] Frank Emmert-Streib Benjamin Haibe-Kains. *Quantitative Assessment and Validation of Network Inference Methods in Bioinformatics.* Frontiers in Genetics, 2015.

[BL14] K. Bache and M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2014.

[BPGC00] H. Bensusan B. Pfahringer and C. Giraud-Carrier. Meta-learning by Landmarking Various Learning Algorithms.. *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 743–750, 2000.

[BPR09] Soares C Brazdil P, Giraud-Carrier C and Vilalta R. *Metalearning: Applications to Data Mining, 1st edition.* Springer-Verlag, Berlin Heidelberg, 2009.

[BS09] Gustavo E.A.P.A. Batista and Diego Furtado Silva. How k-Nearest Neighbor Parameters Affect its Performance. *Simposio Argentino de Inteligencia Artificial*, pages 95–106, 2009.

[CH15] Tianqi Chen and Tong He. Higgs boson discovery with boosted trees. *JMLR: Workshop and Conference Proceedings*, 2015.

[DAG15] J. A. Fischer R. J. Foley R. R. Gupta R. Kessler A. G. Kim R. C. Nichol P. Nugent A. Papadopoulos M. Sako M. Smith M. Sullivan R. C. Thomas W. Wester R. C. Wolf F. B. Abdalla M. Banerji A. Benoit-Lévy E. Bertin D. Brooks A. Carnero Rosell F. J. Castander L. N. da Costa R. Covarrubias D. L. DePoy S. Desai H. T. Diehl P. Doel T. F. Eifler A. Fausti Neto D. A. Finley B. Flaugher P. Fosalba J. Frieman D. Gerdes D. Gruen R. A. Gruendl D. James K. Kuehn N. Kuropatkin O. Lahav T. S. Li M. A. G. Maia M. Makler M. March J. L. Marshall P. Martini K. W. Merritt R. Miquel B. Nord R. Ogando A. A. Plazas A. K. Romer A. Roodman E. Sanchez V. Scarpine M. Schubnell I. Sevilla-Noarbe R. C. Smith M. Soares-Santos F. Sobreira E. Suchyta M. E. C. Swanson G. Tarle J. Thaler A. R. Walker D. A. Goldstein, C. B. D'Andrea. Automated transient identification in the dark energy survey. *The Astronomical Journal*, 150(3), 2015.

[DF13] Ali Jannesari Felix Wolf Daniel Fried, Zhen Li. Predicting Parallelization of Sequential Programs Using Supervised Learning. *Machine Learning and Applications (ICMLA), 2013 12th International Conference*, 2013.

[EOY17] Young Cheol Yoon Dong Wook Kim Sunyoung Kwon Eunsun Oh, Sung Wook Seo and Sungroh Yoon. Prediction of pathologic femoral fractures in patients with lung cancer using machine learning algorithms: Comparison of computed tomographybased radiological features with clinical features versus without clinical features. *Journal of Orthopaedic Surgery*, 25(2):1–7, 2017.

[Est17] Estimation lemma. Bootstrap aggregating — Wikipedia, the free encyclopedia, 2017. [Online; accessed 12-September-2017].

[FDG+12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[FE17] Nicolo Fusi and Huseyn Melih Elibol. Probabilistic Matrix Factorization for Automated Machine Learning. *Cornell university library*, 2017.

[FP01] J. F¨urnkranz and J. Petrak. An evaluation of landmarking variants. *Proceedings of the ECML/PKDD Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, page 57–68, 2001.

[HBK00] Christophe Giraud-Carrier Hilan Bensusan and Claire Kennedy. A higher-order approach to meta-learning. *In Proceedings of the ECML'2000 workshop on Meta-Learning: Building Automatic Advice Strategies for Model Selection and Method Combination*, page 109–117, 2000.

[HLL13] Zhihao Yang Hongfei Lin, Kavishwar B Wagholikar and Hongfang Liu. Identifying protein complexes with fuzzy machine learning model. *Proteome Science*, 2013.

[ISW16] Adam Prugel-Bennettand Iwan Syarif and Gary Wills. SVM Parameter Optimization using Grid Search and Genetic Algorithm to Improve Classification Performance. *TELKOMNIKA (Telecommunication, Computing, Electronics and Control)*, 14(4):1502–1509, 2016.

[JZ16] Matjaž Gams Jernej Zupančič, Damjan Kužnar. Model selection on the jsi grid: Metis use-case. *Slovenian Conference on Artificial Intelligence*, pages 44–47, 2016.

[LKRK13] A. Lipponen, V. Kolehmainen, S. Romakkaniemi, and H. Kokkola. Correction of approximation errors with random forests applied to modelling of cloud droplet formation. *Geoscientific Model Development*, 6(6):2087–2098, 2013.

[LMK11] Ping Li, Joshua L. Moore, and Arnd Christian König. b-bit minwise hashing for large-scale linear SVM. *CoRR*, abs/1105.4385, 2011.

[MA15] Milad Malekipirbazari and Vural Aksakalli. Risk Assessment in Social Lending via Random Forests. *Expert Systems with Applications*, 42(10):4621–4631, 2015.

[MSK05] Pang-Ning Tan Michael Steinbach and Vipin Kumar. *Introduction To Data Mining*. 2005.

[PBGS16] John G. Baker Philip B. Graff, Amy Y. Lien and Takanori Sakamoto. Modeling the SWIFT Bat trigger algorithm with machine learning. *The Astrophysical Journal*, 818(1), 2016.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Ras17] Sebastian Raschka. *Python Machine Learning.* Packt Publishing Ltd, UK, second edition edition, 2017.

[RB17] Jan Kanty Milczek Michał Tadeusiak Robert Bogucki, Jan Lasek. Early warning system for seismic events in coal mines using machine learning. *Proceedings of the Federated Conference on Computer Science and Information Systems*, 8:213–220, 2017.

[Say11] Saed Sayad. *An Introduction to Data Mining.* University of Toronto, Canada, 2011.

[SGH15] Julián Luengo Salvador García and Francisco Herrera. *Data Preprocessing in Data Mining.* Springer, Switzerland, 2015.

[SS07] S. N. Deepa S.N. Sivanandam. *Introduction to Genetic Algorithms.* Springer Publishing Company, 2007.

[SW10] Claude Sammut and Geoffrey I. Webb, editors. *Decision Stump*, pages 262–263. Springer US, Boston, MA, 2010.

[SW17] Ali Akbar Septiandri and Okiriza Wibisono. Detecting spam comments on Indonesia's Instagram posts. *Journal of Physics*, 801(1):1742–6596, 2017.

[ZX17] Kun Fu and Fan Wu Zhibin Xiao, Yang Wang. Identifying Different Transportation Modes from Trajectory Data Using Tree-Based Ensemble Classifiers. *ISPRS International Journal of Geo Information*, 6(2):57, 2017.