Middle East Technical University | Department of Computer Engineering

CNG 409

Introduction to Machine Learning

Fall '2023-2024

Homework 3

Due date: January 02, 2024, 23:55

Objectives

This assignment aims to fulfill the following objectives:

- To familiarize you with the nested cross-validation technique which is crucial for model tuning and model evaluation/comparison, and enable you to gain hands-on experience with it.
- To familiarize you with the ID3 algorithm at the implementation level.
- To familiarize you with several data preprocessing techniques that can be crucial for ML algorithms to work as expected.
- To familiarize you with Scikitlearn's SVM, Decision Tree, and Random Forest models.
- To provide insights on how to compare multiple ML models on a dataset with statistically significant (reliable) results.

The assignment involves both coding and writing a report. You can prepare your report in Latex or Word (the format of the report is up to you). There should be a single report submitted for all parts.

Part 0 - Nested Cross Validation

As explained in the previous assignment, the cross-validation technique is susceptible to yielding optimistic evaluation results when it is applied to a dataset from which a testing part cannot be spared. Even if a test dataset can be formed from given data, when cross-validation is considered, the trained models' final performance may depend on this particular test dataset. A better approach is to test model performances on different test datasets (extracted from the same whole data) and measure average performances. These datasets could be formed randomly but a better approach is to consider nested cross-validation [1,2] where the cross-validation technique is used two times in a nested manner: the outer cross-validation and inner cross-validation. The outer determines the test parts and the training parts for the inner one. (Please recall, the cross-validation technique is applied over the training part to find the best hyperparameter configuration and the final performance of a model with the best hyperparameter configuration is measured on the test part.). To clarify the execution of nested cross-validation, assume that we would like to evaluate the performance of KNN on a given dataset D of a classification task T and we are not provided a separate test dataset for T (so all we have as a dataset is D, we have to both perform hyperparameter tuning and

model evaluation on D). The outer cross-validation loop partitions D into multiple parts (let's assume, the partitions D_1, D_2, D_3, D_4, D_5) and iterates through these partitions at each step. A particular partition is considered an outer test part (say D_1) and the remaining partitions (D_2, D_3, D_4, D_5) are considered to be an outer training part. With the outer training part, the inner cross-validation is run to find the best hyperparameter setting. The outer training part (combination of D_2, D_3, D_4, D_5) is further divided into partitions (say $P_1, P_2...P_{10}$) and the cross-validation procedure is executed. For each KNN hyperparameter configuration (K, instance-instance distance metric), each P_i is considered to be an inner test dataset and the rest of P_is are the inner training dataset (KNN is run over inner training datasets and tested on inner test datasets). After running, for each hyperparameter configuration, 10 performance evaluation scores are obtained (out of 10 inner test datasets). By considering these scores, the best hyperparameter configuration can be picked. Later KNN is executed with the best hyperparameter setting on the outer training part (the combination of D_2, D_3, D_4, D_5) and its final performance is measured on D_1. For the next outer cross-validation loop, all these steps are run from scratch (D_2 is considered to be the outer test dataset and the combination of D_1, D_3, D_4, and D_5 is the next outer training part for the inner cross-validation). Once the outer cross-validation completes, we obtain 5 different evaluation scores for KNN and the average of these provides more statistically reliable results than the case where we were to form a single dataset out of D. Since at each outer cross-validation step, the inner cross-validation is subject to different outer training parts, we expect to get different best hyperparameter configurations. If the KNN model is to be employed for future data instances, KNN can be executed on D with the best hyperparameter configuration whose average performance score is the highest on the outer test datasets. Further, nested cross-validation could be utilized for comparing different models (ML methods) on D as well. The same steps should be repeated for other models and the model that attains the highest average score on the outer datasets can be regarded as the winner. The most important point is to make sure that all models are trained with the same data points so the models can be trained in the inner-cross-validation loop sequentially.

Similar to naive cross-validation, the class imbalance problem and the arguments (e.g. results being dependent on a particular partitioning) regarding the multiple runs of the cross-validation procedure are valid for nested-cross validation. The outer and inner cross-validations can utilize the stratification technique and they can be executed multiple times by shuffling their corresponding training parts (the outer cross-validation loop shuffles the whole data at each run, the inner cross-validation shuffles its inner training part for each run of it). With these considerations, the computation cost of applying cross-validation (especially, nested cross-validation) becomes significant. But, if computationally possible, this burden should be taken on to be able to report more statistically reliable results.

In summary, if there is plenty of data to form training, validation, and test datasets, a given model can be trained on the training dataset, its best hyperparameter configuration could be found with the validation dataset and its expected generalization performance could be measured on the test dataset (similarly different ML algorithms can be compared concerning their performance on the test dataset). But for the cases where a validation dataset cannot be afforded (a training dataset and a test dataset are available), the cross-validation technique should be considered for training a model and tuning its hyperparameters. After the cross-validation procedure, the model with its best hyperparameter configuration can be trained on the training dataset and its expected generalization performance could be measured on the test dataset (similarly different ML algorithms can be compared depending on their performance on the test dataset). Finally, when both validation and test datasets cannot be afforded, the nested cross-validation technique should be considered for training, tuning, and evaluating a model (in order to compare different methods/algorithms as well). The very basic idea of the nested cross-validation procedure is to find the optimal hyperparameter values by considering as many train-test dataset pairs as possible (in the inner cross-validation loop) and test/evaluate the model with the best hyperparameter values found on as many test-training dataset pairs as possible (in the outer cross-validation loop) in order to be able to obtain statistically significant/reliable results for model evaluation, model tuning.

Note 1: When the model to be trained/tuned/evaluated involves any randomization, the model should be trained and tested multiple times to alleviate the effects of randomization. For instance, a feed-forward

neural network (MLP) requires its weights to be initialized randomly. When there are training, validation, and test datasets, the MLP model should be trained and tuned on the training and validation datasets multiple times to find the best hyperparameter setting (say 10 times for each hyperparameter configuration, depending on the 10 results attained over the validation dataset, the best hyperparameter values can be determined). After determining the best hyperparameter configuration, the model should be trained on the training dataset (possibly by merging it with the validation dataset) multiple times and its performance should be measured for each run (confidence intervals should be provided to bolster that the results are not a fluke). The same considerations are valid for cross-validation and nested cross-validation. For a particular training part and test part, the model should be run multiple times and results should be averaged accordingly for finding the best hyperparameter configuration. Similarly, the model should be trained multiple times while measuring its expected generalization performance on the test dataset.
Note 2: Application of nested cross-validation with a method that features some randomization becomes challenging with respect to computation cost (especially when both inner and outer cross-validation loops are repeated multiple times in addition to executing the model multiple times with a particular training and testing parts). This strategy should be considered for attaining statistically reliable results when applicable. But this strategy quickly becomes infeasible especially when the model is complex (e.g. a neural network with a high number of hidden layers and hidden layer units, a single training execution may take a significant amount of time to complete.)

Part 1 - Decision Trees and Random Forests

Decision trees are non-parametric models that partition each instance feature dimension into finite regions and decision boundaries are formed in such a way that instances are confined into subspaces formed by these regions. Each level in the tree represents a different feature dimension and each dimension partitioning is represented by branches originating from a particular tree node. For a given set of data instances, many decision trees can be formed but the one that has the smallest structure is preferred over the others due to computation demands. Since finding an optimal decision tree in size is an NP-Complete problem [3], approximate optimal structures are formed via heuristic functions (e.g entropy, gini index) that aim to shrink the tree size. Since while yielding a result (e.g. classifying an instance), the path taken by the tree can be tracked and all the decisions made can be represented as if-else rules. Hence decision trees facilitate interpretability.

Suppose we have a regression model $f(x; D)$ trained on a dataset D ($D = (x_i, y_i)|i = 0, 1....N$) that has been generated by a function $F(x)$ ($y_i = F(x_i)$) which is unknown. In order to measure how well the regression model approximates/estimates the actual data-generating function, the expected squared error between the estimator and the actual generation function can be calculated [4]:

$$\mathbb{E}_D[f(x; D) - F(x)^2]$$

The expectation is taken over all possible datasets (all D_is, for a particular D_i $f(x; D_i)$ can approximate F(x) very well but some other D_j may fail to do so, so we need to calculate the average expected difference between the actual generating function and the estimated function (by the model) over all possible datasets). This expectation could further be written as [4]:

$$\mathbb{E}_D[f(x; D) - F(x)^2] = (\mathbb{E}_D[f(x; D) - F(x)])^2 + \mathbb{E}_D[(f(x; D) - \mathbb{E}_D[f(x; D)])^2]$$

where the first term on the right-hand side is coined as $Bias^2$ ($Bias = \mathbb{E}_D[f(x; D) - F(x)]$ and the second term is coined as $Variance$ [4, 5]. $Bias$ measures how much the estimated model deviates from the actual data-generating function. $Variance$ measures the variance (variability) of the estimator (the regression model f) (it calculates how much the estimated model deviates from its expected value (mean)). This decomposition reveals what is known as the bias-variance dilemma [5,6] (overfitting and underfitting concepts can easily be expressed with this decomposition) (variance-bias dilemma is valid for classification

as well [4,5]). The model trained has to balance between bias and variance terms in order to lower the expected squared error. A complex model can keep the bias term very small but due to its high flexibility, it can easily lead to a high variance term (overfitting, since the model is too flexible, with a given dataset it can significantly deviate from its expected function value). On the contrary, a simple model is likely to attain a very low variance term but its bias term is likely to be high because it can easily fail to approximate the actual data-generating function (due to its simplicity) (underfitting). One important strategy to reduce the variance term is to employ multiple instances of the model (also different models can be considered) and utilize all of them during task learning [7] (another important strategy to reduce variance is to utilize regularization techniques [5]). This strategy is broadly coined as ensemble learning [7,8] and it is the basic approach considered by random forest models [9]. A random forest model trains multiple decision trees by incorporating randomization into the learning procedure to increase the variability of the trained decision trees hence it aims to lower the model variability (hence mitigate overfitting) by employing all these trained trees for a given task [9,10,11]. [9] proposes choosing a subset of features randomly while splitting at a node and forming new features by combining some or all of the existing features randomly to create multiple diverse decision trees. The Scikitlearn implementation of a random forest model [10,11] builds decision trees by sampling data instances randomly from a given training dataset and considering all features or a random subset of them for splitting at a node. Since multiple decision trees are constructed with considerable variability (variance), their ensemble model (random forest) is expected to be more robust to outliers and noise [9], which can lead to better generalization.

The ensemble of decision trees is not limited to random forests. Generic ensemble methods (generic in the sense that in addition to decision trees, they can be utilized with other machine learning methods as well) such as bagging [8] and boosting [8] could be employed with decision trees to benefit from the advantages of ensemble learning. The bagging approach combines/aggregates predictions of multiple ML algorithms by training them on random subsets of a given dataset. On the other hand, the boosting method trains ML models in steps. A new model is trained depending on the mistakes made by previously trained models. For instance, AdaBoost [8] initially selects data instances randomly and trains an ML model. After training, the algorithm assigns weights to the data instances based on the success of the trained model. These weights determine the probability of the data instances being selected for the next training step. Correctly predicted data instances get low weight values, whereas falsely predicted data instances get assigned to higher weight values. Thus, adaptively, models are trained by focusing on the data instances that are harder to learn. Similarly, gradient boosting [12] uses the gradient of an error function between two training steps to train models. When decision trees are employed, a decision tree is trained in each boosting step (e.g., gradient-boosted decision trees [13]).

Part 1 Specifications

- For this part, you are expected to implement the ID3 algorithm (you can refer to lecture materials for the pseudo-code of the algorithm) on a given dataset without using any library implementation.
- Your are provided with 2 Python files: `ID3.py` and `mainID3.py`.
- `mainID3.py` defines a small dataset on which a decision tree model defined in `ID3.py` is trained.
- You are expected to complete the functions of `DecisionTree` class defined in `ID3.py`.

```
# In the decision tree, non-leaf nodes are going to be represented via TreeNode
class TreeNode:
    def __init__(self, attribute):
        self.attribute = attribute
        # dictionary, k: subtree, key (k) an attribute value, value is either TreeNode or TreeLeafNode
        self.subtrees = {}

```

```python
# In the decision tree, leaf nodes are going to be represented via TreeLeafNode
class TreeLeafNode:
    def __init__(self, data, label):
        self.data = data
        self.labels = label

class DecisionTree:
    def __init__(self, dataset: list, labels, features, criterion="information gain"):
    def calculate_entropy__(self, dataset, labels):
    def calculate_average_entropy__(self, dataset, labels, attribute):
    def calculate_information_gain__(self, dataset, labels, attribute):
    def calculate_intrinsic_information__(self, dataset, labels, attribute):
    def calculate_gain_ratio__(self, dataset, labels, attribute):
    def ID3__(self, dataset, labels, used_attributes):
    def predict(self, x):
```

Depending on the parameter `criterion` (of type string, either "information gain" or "gain ratio") of `DecisionTree`, the split decisions should be made with the information gain score or gain ratio score, respectively.

- For the decision tree implementation of this part, you don't need to consider any pruning method.

Part 2 - Data Preprocessing, SVM, Kernel Functions

Depending on task and data generation processes, dataset instances may contain a different number of features with different value types (e.g. real number, integer, boolean, string, .etc) and different value ranges. Some features may take values from the real number set ($\mathbb{R}$) or integer set ($\mathbb{Z}$), some may take values from a finite set of integers or some may even have non-numerical values (e.g. boolean, string). Furthermore, numerical features may have different value ranges. For instance, a feature that represents age information can have values from the range [0, 150] or the feature for yearly income may take values from [1000, 100000]. Different value ranges are of great importance while applying an ML algorithm on a dataset because some features can easily dominate loss function calculations (some can contribute more than others to the loss value) or features with inappropriate values may hinder an algorithm to learn as expected (e.g. with huge input values, MLPs may not learn well since the activation function used (e.g. sigmoid) can easily saturate and gradient magnitudes may become too small for them hence the loss function may not be optimized as expected. A similar argument can be made for SVM because after learning SVM weights with different input ranges, some weights of SVM can become too large and they can easily dominate while producing a prediction result compared to other feature weights. With respect to the task, other features may be of great importance for the final decision but since some weight values dominate theirs, their contribution to the final decision may become negligible thus the trained SVM model learns an incorrect weighting scheme. (after learning weight values, by inspecting them we can determine which features are more important for the decision since a decision is made by simply multiplying the weight vector with a given data instance vector. In order for this interpretation strategy to work all input values should be in the same value range)).

Having to work with different types of feature values and different feature value ranges necessitates what is coined as data preprocessing. If parametric methods (such as MLP, SVM, Logistic regression, .etc) are to be employed, vectors for data instances should be formed (even for non-numerical features if they are not eliminated for the task). For some non-parametric methods (such as KNN) too, input vectors should be formed (as a non-parametric method, a decision tree does not necessarily require input vectors to be formed for a given dataset since it works with individual features. It also does not

require all feature values to be in the same value range (features can have different value types and value ranges)). After forming vectors, the feature values should be modified to alleviate/eliminate the problems that have been mentioned previously. To this end, the following preprocessing operations could be considered for modifying feature values. Suppose that we have the training, validation, and test datasets ($X^{\text{train}}, X^{\text{validation}}, X^{\text{test}}$), and each data is represented with row vectors (Xs are matrices each row of which represents a data instance):

$$\hat{x}_{ij}^{train} = \frac{x_{ij}^{train} - \mu_j^{\text{train}}}{\sigma_j^{\text{train}}}$$

$$\hat{x}_{ij}^{validation} = \frac{x_{ij}^{validation} - \mu_j^{\text{train}}}{\sigma_j^{\text{train}}}$$

$$\hat{x}_{ij}^{test} = \frac{x_{ij}^{test} - \mu_j^{\text{train}}}{\sigma_j^{\text{train}}}$$

where $\mu_j^{\text{train}} = \frac{\sum_{i=1}^{N^{\text{train}}} x_{ij}^{\text{train}}}{N^{\text{train}}}, \sigma_j = \sqrt{\frac{\sum_{i=1}^{N^{\text{train}}} (x_{ij}^{\text{train}} - \mu_j^{\text{train}})^2}{N^{\text{train}}}}, N^{\text{train}}$ is the number of instance in the training dataset and x_{ij}s are j^{th} feature value of i^{th} data instance of training, validation or test datasets ($\hat{x}_{ij}$ is the new feature value). This type of modification is coined as standardization [14] since it aims to have feature values with a mean value of zero and unit variance ($\sigma_j = 1$). Another technique which is known as min-max normalization [14,15] transforms each feature value into a value in the range $[a, b]$:

$$\hat{x}_{ij}^{train} = a + \frac{(x_{ij}^{\text{train}} - \min_l x_{lj}^{\text{train}})(b-a)}{\max_k x_{kj}^{\text{train}} - \min_l x_{lj}^{\text{train}}}$$

$$\hat{x}_{ij}^{validation} = a + \frac{(x_{ij}^{\text{validation}} - \min_l x_{lj}^{\text{train}})(b-a)}{\max_k x_{kj}^{\text{train}} - \min_l x_{lj}^{\text{train}}}$$

$$\hat{x}_{ij}^{test} = a + \frac{(x_{ij}^{\text{test}} - \min_l x_{lj}^{\text{train}})(b-a)}{\max_k x_{kj}^{\text{train}} - \min_l x_{lj}^{\text{train}}}$$

All validation and test data instances are supposed to be transformed with the statics (e.g. min, max, mean) of the training dataset in order to avoid data leakage [16] (no data instances from validation or test dataset should be for preprocessing, preprocessing statistics should only be calculated out of the training dataset. Otherwise, some information from the test dataset will be incorporated/leaked into the preprocessing step). As for converting non-numerical feature values into numeric vectors, the one-hot encoding technique can be considered. For instance, suppose that we have 3 unique values for a feature: "student", "instructor", "visitor", and with this technique, these feature values can be converted to the following vectors: [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0], respectively. To form a data instance vector, these generated one-hot encoded vectors can be merged with the vectors formed from numerical features.

During the training of an SVM model, the following dual form needs to be maximized (for both hard-margin and soft-margin SVM models) [17]:

$$L_d = -\frac{1}{2} \sum_t \sum_s \alpha^t \alpha^s r^t r^s (x^t)^T x^s + \sum_t \alpha^t$$

where α^t and α^s are Lagrange multipliers for instance t and s, r^t and r^s are instance labels of instance t and s, x^t and x^s are instance vectors of instance t and s, respectively. At the heart of the function to be maximized, there lies a dot product operation, namely $(x^t)^T x^s$, no matter what the size of instance vectors

is (the dot product operation yields a single numeric value). In a sense, this dot product operation can be considered as a similarity/dissimilarity measure between two instances (it is the unnormalized cosine distance).

Thanks to Cover's theorem [18], probabilistically, non-linearly separable data instances in a d-dimensional space may become linearly separable when they are non-linearly transformed/mapped into a higher dimensional space than d [19]. Let ϕ be a non-linear mapping and $\bar{x}$ be the mapped version of a vector x ($\bar{x} = \phi(x)$, $\bar{x}$ is in a higher dimensional space than x). Being backed up by Cover's theorem, we could first map (non-linearly) our data instances of a given task and train an SVM model on these mapped data instances in a higher dimensional space (in the hope that newly mapped instances are linearly separable in the new dimensional space). With this dimension changing trick, the dual form becomes:

$$L_d = -\frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s (\phi(x^t))^T \phi(x^s) + \sum_t \alpha^t$$

In order to calculate the new dot product, we need to first map data instances to the new space. To get rid of this computation overhead, the dot product operation can be represented with a function K ($K(x^t, x^s) = \phi(x^t)^T \phi(x^s)$):

$$L_d = -\frac{1}{2}\sum_t \sum_s \alpha^t \alpha^s r^t r^s K(x^t, x^s) + \sum_t \alpha^t$$

With this function trick, instance-instance similarities can be calculated without actually transforming data instances into a new space. The family of all suitable functions (Ks) is coined kernel functions [17]. The following are several examples of this family [17]:

$$K(x^t, x^s) = \mathrm{e}^{-\frac{\|x^t - x^s\|^2}{2\sigma^2}}, \text{Gaussian Radial Basis Function[20]}$$

$$K(x^t, x^s) = tanh(2(x^s)^T x^t + 1), \text{Sigmodial[17]}$$

Note 1: After a non-linear mapping into a higher dimensional space, data instances are not guaranteed to become linearly separable. Cover's theorem states that they become more likely to be linearly separable [19].
Note 2: For a given classification task, after training, at the output a Multilayer Perceptron Network (MLP) data instances become linearly separable at a d dimensional space where d is the number of output nodes since the output is computed with a weighted average of the output of the previous layer (the last layer defines linear decision boundaries, softmax is used to obtain label probabilities). An MLP with one or more hidden layers can be considered to map its input data non-linearly into spaces whose dimensions are defined by the number of nodes in the hidden layers so as to render them linearly separable and all these non-linear mappings at the hidden layers are learned by the network itself via the backpropagation algorithm.
Note 3: Preprocessing a test dataset with the statistics calculated on a training dataset is valid for the cross-validation procedure too. If the data needs to be normalized, statistics should be extracted from training partitions and the test partition should be processed depending on these statistics (this preprocessing should be executed from scratch for each test-training partitioning).

Part 2 Specifications

- For this part, you are given two datasets (`dataset1` and `dataset2`) and two partially implemented Python files (`svm_dataset1.py`, `svm_dataset2.py`, they load their corresponding datasets).
- On `dataset1`, you are expected to run an SVM model of Scikitlearn (SVC [21]) and plot the trained model's decision boundaries along with the data points. You are expected to try at least 4 configurations (2 values for the C parameter and 2 values for the kernel function parameter, values to try

are up to you) and provide each configuration's resulting plots in your report (4 plots with decision boundaries of the SVM model). All your implementation should be provided in `svm_dataset1.py` and you don't need to perform the cross-validation procedure and data preprocessing for this task.

- On `dataset2`, you are expected to perform cross-validation to find the best hyperparameter values and provide confidence intervals for each hyperparameter as a table in your report by repeating the cross-validation procedure 5 times (by shuffling the dataset each time, 10 splits). You are expected to test at least 4 configurations (values to try are up to you but you should try at least two kernel functions). All your implementation for this task should be provided in `svm_dataset2.py`. Scikitlearn provides a very versatile class (**GridSearchCV** [22]) to combine grid-search with the cross-validation procedure. You can use it for this task.

- For `dataset2`, please preprocess the data by using **StandardScaler** [23] of Scikitlearn within the cross-validation loop. The test partition is expected to be processed via the statistics calculated from the training partitions.

- You can consider any performance metric (accuracy, precision, recall) while determining the best hyperparameter values on `dataset2`.

- You are expected to utilize the whole dataset for the cross-validation procedure. You can assume that the test dataset is going to be given later for the final evaluation. Please in your report provide all cross-validation procedure results for hyperparameter values as a table.

- For this part, you can benefit from the source codes provided by the Scikitlearn documentation.

Part 3 - Method Comparison

No free lunch theorem [24,25,26,27] states that when all possible optimization problems are considered (under some conditions) two (optimization) algorithms attain the same average performance and this is mathematically stated as:

$$\sum_f P(d_m^y | f, m, a1) = \sum_f P(d_m^y | f, m, a_2)$$

where a_1 and a_2 are optimization algorithms, f is the optimization problem (cost function), d_m^y is the ordered set of cost values, $P(d_m^y | f, m, a_i)$ is the conditional probability of obtaining a sequence of cost values (d_m^y) by running the algorithm a_i on the problem (f) m steps. So there is no single optimization algorithm that outperforms the random search when all possible optimization problems are concerned. If an algorithm performs better on a family of particular problems, it has to perform worse on the remaining problems (it pays for its performance gain on the remaining problems, no free lunch).

When considered within the machine learning context, learning is realized by optimizing performance/-cost functions and no free lunch theorem is valid for ML methods/algorithms [26,27] (they aim to minimize/maximize an objective function and optimized functions change as datasets change, loss functions generate different values on different datasets hence we can think that every problem with a different dataset has a different loss function). Hence we cannot expect to have an ultimate ML algorithm that outperforms all the other methods when we consider all possible ML problems (on average it is no better than an algorithm that generates random results). This implies that rather than looking for the best algorithm for all problems, we should focus on searching for the best algorithm for a particular problem family (on a subset of all problems an algorithm may outperform the others). This search endeavor can be carried out by testing all suitable methods for a particular type of problem along with devising/proposing new algorithms, which is the basic reason for ML research should continue.

In this part, you are expected to compare several algorithms on two datasets with the nested cross-validation procedure.

Part 3 Specifications

- For this part, you are given a dataset and expected to compare 4 algorithms (KNN, SVM, Decision Tree, and Random Forest) on it using the nested cross-validation technique.
- The dataset of this part is a well-known credit application dataset [28] where credit applicants are represented with 20-D vectors with both numeric and categorical feature values. The task involves classifying applicants into 2 categories (good or bad) for credit application.
- For data loading, you are given a Python file: `part3.py`. Further, you can inspect the data loading function to gain some insights into how categorical and numeric features can be combined together for parametric models.
- For each method please make sure that at least 2 hyperparameter configurations are tested (the hyperparameter values to test are up to you. Due to time constraints, the number of configurations has been kept small but if you would like you can try more configurations to attain more reliable comparison results.)
- Please employ the min-max normalization transformation (all feature values should be in the range [-1, 1]) during the inner-cross-validation loop. Please make sure that the test partition is normalized depending on the statistics coming from the training partitions.
- Run all the methods on these preprocessed data instance vectors.
- During nested cross-validation, please make sure that all methods are subject to the same training and test partitionings. In the inner-cross-validation loop, you can train them sequentially.
- Since the random forest model is a stochastic model, in the inner-cross-validation loop please make sure that you have run it at least 5 times on a particular test and training partitioning to get more reliable results. This consideration is not valid for the other methods (since they do not involve any randomization that will significantly alter the course of learning. (SVM involves but it deals with a convex loss function and converges almost the same point in each run.))
- Please compare all methods with respect to their accuracy and F_1 scores separately (along with their confidence intervals).
- Please in your report explain your experimental setup and please provide results in tables with confidence intervals. The tables should contain both hyperparameter search results and final evaluation results.
- Please for the outer cross-validation loop consider **RepeatedStratifiedKFold** [29] (3 splits, 5 repetitions with shuffle) and for the inner loop again **RepeatedStratifiedKFold** (5 splits, 5 repetitions with shuffle). For the hyperparameter search, you can utilize **GridSearchCV** in the inner cross-validation loop.
- For this part, you can benefit from the source codes provided by the Scikitlearn documentation.
- Please provide all your implementation of this part in `part3.py`.
- If you would like, you can incorporate other methods into the experiments, such as MLPClassifier [31] and GradientBoostingClassifier [30] of Scikitlearn. Since randomization is essential for both methods, they should be treated similarly to random forests in experiments.
- As in a decision tree, features are ordered/layered depending on their importance depending on a criterion (e.g., gini index, entropy). After training, we can check which features have been the most influential in learning a dataset. At the end of `part3.py`, please load the credit application

dataset without converting discrete feature values into one-hot vector representation and train a DecisionTreeClassifier [32]. After training, in your report, please state which five features (you can refer to feature explanations in the data loading function) have been important for classifying data instances in the credit application dataset.

- Similarly, please train an SVC [21] by loading the credit application dataset via one-hot vector representations for discrete features and provide support vectors (both for positive class and negative class separately) in your report. Also, in your report, please briefly comment on similarities or commonalities of the support vectors of each class by inspecting feature values. You can add the implementation for this part at the end of `part3.py` too.

Regulations

1. You are expected to write your code in Python by using scikitlearn, Numpy, copy, and Matplotlib libraries.

2. Falsifying results or changing the composition of training, validation, and test data is strictly forbidden, and you will receive 0 if this is the case. Your programs will be examined to see if you have actually reached the results and if it is working correctly.

3. **Commenting:** Since all implementations are going to be inspected manually, comments are of great importance for the evaluation procedure. Please add extensive explanatory comments in all of your implementations.

4. **Late Submission:** You have a total of 5 late days for all homework without receiving a penalty. As soon as you have depleted your quota, penalization will be in effect. The late submission penalty will be calculated using $5d^2$, that is, 1 day late submission will cost you 5 points, 2 days will cost you 20 points, and 3 days will cost you 45 points. No late submission is accepted after reaching a total of 3 late days (No matter whether you have still a remaining late-day quota or not).

5. **Cheating:** Using any piece of code that is not your own is strictly forbidden and constitutes cheating. This includes friends, previous homework, or the internet. However, example code snippets shared on Scikitlearn's website can be used. **We have a zero-tolerance policy for cheating**. People involved in cheating will be punished according to university regulations.

6. **Discussion:** You must follow ODTUClass for discussions and possible updates/corrections/clarifications on a daily basis. **For the first and previous assignments, we received many questions that could have been asked on the discussion forum. Please ask your questions on ODTUClass unless you really think that your question is private and does not concern anyone**.

7. **Evaluation:** Your assignment is going to be graded manually.

Submission

Submission will be done via the ODTUClass system. For all parts, you are expected to upload a single pdf file named **report.pdf**. For Part 1, you are expected to upload **ID3.py** file. For Part 2 you are expected to upload **svm_dataset1.py** and **svm_dataset2.py** files. For Part 3, you are expected to upload **part3.py**.

References

1. https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html
2. https://machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/
3. Hyafil L., Rivest R. L. (1976). Constructing optimal binary decision trees is np-complete. Information Processing Letters, 5(1), 15–17.
4. Richard O. Duda, Peter E. Hart, and David G. Stork. 2000. Pattern Classification (2nd Edition) Chapter 9.
5. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 4.
6. https://en.wikipedia.org/wiki/Bias-variance_tradeoff
7. Zhang, Cha, and Yunqian Ma, eds. Ensemble machine learning: methods and applications. Springer Science and Business Media, 2012 Chapter 1.
8. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 17.
9. Breiman, L. (2001). Random Forests. Machine Learning 2001 45:1, 45(1), 5–32.
10. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
11. https://scikit-learn.org/stable/modules/ensemble.html#forest
12. Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine.." Ann. Statist. 29 (5) 1189 - 1232, October 2001. https://doi.org/10.1214/aos/1013203451
13. https://scikit-learn.org/stable/modules/ensemble.html
14. https://en.wikipedia.org/wiki/Feature_scaling
15. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html #sklearn.preprocessing.MinMaxScaler
16. https://scikit-learn.org/stable/common_pitfalls.html
17. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 13.
18. Cover, T.M. (1965). "Geometrical and Statistical properties of systems of linear inequalities with applications in pattern recognition" (PDF). IEEE Transactions on Electronic Computers. EC-14 (3): 326–334.
19. Neural Networks and Learning Machines, 3rd edition, Simon Haykin, Chapter 5.
20. https://en.wikipedia.org/wiki/Radial_basis_function
21. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
22. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
23. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
24. Wolpert, D. H., and Macready, W. G. (1996). No Free Lunch Theorems for Optimization.
25. https://en.wikipedia.org/wiki/No_free_lunch_theorem
26. Introduction to Machine Learning, 2nd edition, Ethem Alpaydın, Chapter 19.

27. https://machinelearningmastery.com/no-free-lunch-theorem-for-machine-learning/

28. https://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data)

29. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratifiedKFold.html

30. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

31. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

32. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html #sklearn.tree.DecisionTreeClassifier

33. Lecture notes

34. Announcements Page

35. Discussion Forum