Exercise 1: Classification

Group 50: Sebastian Lindner (00571432), Daniel Zabielski (12119018), Yahya Jabary (11912007)

Introduction

This report presents a classification exercise comparing 3 models across 4 diverse datasets.

The goal was to examine the impact of dataset size, preprocessing techniques, and hyperparameter tuning on classifier performance. Results were analyzed to identify trends and draw insights into factors driving classification performance (understood as efficiency, accuracy and robustness) across different scenarios.

Emphasis is placed on the experimental methodology, providing a framework for comparing classifiers and understanding their behavior under different scenarios.

Obviously there is no one-size-fits-all solution, but the insights gained from this exercise can help guide future decision-making when selecting classifiers and preprocessing techniques for new datasets (in a similar vein to the "No Free Lunch Theorem"[1] in machine learning).

1. What datasets did we choose and why?

We selected 4 datasets for this exercise, 2 of which were from a previous exercise and 2 from a Kaggle competition. The datasets were chosen to be diverse in terms of size, number of features and number of classes. This diversity was intended to provide a broad range of scenarios for testing the classifiers.

We gave each of the datasets a code name for easier reference (you will see these names throughout the report):

- `congress`: Congressional Voting Records

  Source: https://archive.ics.uci.edu/dataset/105/congressional+voting+records

  This dataset tracks how members of the US House of Representatives voted on 16 important issues. The votes were identified by the Congressional Quarterly Almanac (CQA) during the 98th Congress, 2nd session in 1984. It shows if they voted yes ("y"), no ("n") or didn't express a clear opinion ("unknown"). The dataset also notes each representative's political party (Democrat or Republican). The issues covered in the dataset include things like aid for infants with disabilities, water project funding and foreign policy matters.

  The dataset has 218 samples and 15 features (issues) and 1 target variable (party affiliation / "class").

  All classes except for the "class" / party affililation column which is "Democrat" or "Republican" are either "y", "n" or "unknown".

  The goal is to determine the party affiliation of a representative based on their voting record. Notably 61% of the samples are Democrats, while 39% are Republicans - the effects of this proportion on the target value throughout history could be interesting to explore through time series analysis.

  Since this is a classic binary classification problem intuitively one would expect this dataset to be well-suited for a decision tree classifier, as the voting patterns of the representatives could be easily separated by a series of yes/no questions - but we will see if this holds true in practice.

[1] Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE transactions on evolutionary computation, 1(1), 67-82.

The data is well-structured and has a lot of potential for optimization by ie. using one-hot encoding for the categorical features to speed up the training process.

- `mushroom`: Secondary Mushroom

  Source: https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset

  The Secondary Mushroom dataset is inspired by the original Mushroom dataset[2] from 1987. The original dataset contains *hypothetical* information about mushrooms and whether they are edible or poisonous. The features include things like cap shape, cap color, gill color, stalk shape, etc.

  It was created in 2023 and is intended for binary classification into edible and poisonous mushrooms. It contains 61,068 instances of *hypothetical* mushrooms with 20 features, including variables like cap diameter, shape, surface, color, and others related to mushroom characteristics. The dataset was simulated based on the Mushroom Data Set by J. Schlimmer. The dataset was created as a product of a bachelor thesis at Philipps-University Marburg.

  It has a mix of categorical and numerical features, which could make this dataset more challenging to work with, combined with the larger quantity of samples. A performance improvement could maybe be achieved by finding the features with the strongest correlation to the target variable and focusing on them to reduce the dimensionality of the dataset.

  We chose this dataset to see how the classifiers perform on a larger dataset with a mix of categorical and numerical features.

- `reviews`: Amazon Reviews

  Source 1: https://www.kaggle.com/competitions/mse-bb-2-ss2020-mle-amazon-reviews

  Source 2: https://www.kaggle.com/competitions/184702-tu-ml-ws-19-amazon-commerce-reviews/

  Source 3: https://www.kaggle.com/code/anybodywise/yavorskyi-predict-review-authors-with-sklearn/

  The origin of this dataset isn't entirely clear. It seems to be a collection of Amazon reviews, purely based on the available file names. But upon further investigation, we found out that the dataset was used in multiple Viennese university courses and kaggle competitions that provide some context: This dataset originates from customer reviews on the Amazon Commerce Website and is used for authorship identification. Previous research typically focused on experiments with two to ten authors - but this adaption consists of 50 highly engaged users, each identified by a unique username, known for frequently posting reviews in these forums.

  The authors are identified by their unique username, but the review metadata is anonymized, which makes it almost impossible to infer the author from the review content or gain any insights based on intuition / domain knowledge.

  The dataset contains 1,000 features and 750 samples. All features are numerical and the target variable is the author of the review. The dataset is well-structured but has an incredibly high dimensionality, which could make it challenging to work with. The dataset is also imbalanced, with some authors having more reviews than others.

- `seattle`: Seattle Crime 6

  Source: https://www.openml.org/search?type=data&id=41960&sort=runs&status=active

[2]Original Mushroom dataset: https://archive.ics.uci.edu/dataset/73/mushroom

The last dataset contains various information about criminal offences in Seattle, WA. We mainly chose this dataset because of its difference in characteristics to the first one and its similarly interesting content and good presentation.

It has 523,590 samples and 9 attributes, where the "Occurred_Time", "Reported_Time", "Crime_Subcategory", "Precinct" and "Sector" attributes have missing values. Here we have identified 1 target variable we want to focus on, which is the "Primary_Offense_Description", a nominal quantity, which can assume 144 distinct values, one for each offense.

Concerning pre-processing, individual values in the target attribute could be condensed to a few groups containing multiple felonies to make the actual classification task easier. Additionally, the missing values could be filtered out or filled in.

Anomaly:

On a side note, we encountered some anomalies with the OpenML platform while accessing their data API. The `Report_Number` column was missing from the datasets we fetched via the API, but present in the datasets we manually downloaded from the website. This inconsistency could cause problems for users and should be addressed by the platform as it could lead to irreproducible results. Furthermore, we found that offline datasets from OpenML are in binary format and need to be decoded, unlike the API versions.

Also, surprisingly, reading the offline datasets was slower than reading them via the API even though we expected the opposite before benchmarking the runtime. The unreliable performance of the OpenML API could be a concern for users who rely on it for their research.

2. How did we find the best model training approach for each dataset?

The methodology that will be presented provides a solid starting point for exploring the relationships between datasets, preprocessing techniques, machine learning models, and their corresponding hyperparameters and validation strategies.

The core strength of our approach lies in its systematic exploration of various combinations of these elements through grid search. By testing different techniques for each dataset, the methodology aims to provide an initial understanding of which configurations might perform well on each dataset, before diving deeper into hyperparameter tuning and performance evaluation and more rigorous comparisons, allowing one to determine whether the differences in performance between various combinations are statistically significant.

2.1. Choosing classifiers, preprocessing techniques and metrics for grid search

We started by defining a set of configurations to test the impact of different preprocessing techniques and classifiers on the datasets. The configurations were chosen to cover a wide range of possibilities, including different scaling methods, strategies for handling categorical data, imputation methods, classifier types, and validation techniques. Before delving into the concrete strategy, however, our modus operandi concerning preprocessing, classifier and metric selection will be explained.

2.1.1 Preprocessing

The use of preprocessing techniques such as scaling and imputation demonstrates the impact of data makeup and completeness on model performance, particularly as machine learning algorithms can be sensitive to feature ranges and missing values. Accordingly, we chose to apply preprocessing techniques everywhere it made sense in order to supply our models with data they can easily process.

1. `congress`: As has already been mentioned, this dataset only contains categorical features, so any numerical preprocessing strategies can be disregarded. Due to its entirely categorical nature, labelling and one-hot-encoding are strong candidates for preprocessin, though. Since it also contains missing values, imputation and/or deletion of rows/columns must be considered. Here, we decided to only go with imputation and its associated methods (impute with mode or make "missing" its own new value), as the "unknown"s appear too often and are spread too consistently across the dataset, making deletions of any kind infeasible.
2. `mushroom`: This dataset contains both numerical and categorical features, which leaves all preprocessing strategies on the table. For the numerical features, we considered the canon "normal" and "minmax" scaling options. For the categorical data, we went with the usual labelling and one-hot-encoding. Concerning missing values, we again decided that they were spread too consistently across the dataset to make deletion an option, so we went with our already established imputation techniques.
3. `reviews`: This dataset contains only numerical features and no categorical ones. Additionally, it has no missing values, leaving only scaling as a possible preprocessing strategy.
4. `seattle`: Similarly to `mushrooms`, this dataset contains both numerical and categorical features as well as missing values. Accordingly, all of the same preprocessing strategies mentioned for the `mushrooms` dataset can be applied here as well.

2.1.2 Classifiers and validation techniques

- The selection of classifiers—Multilayer Perceptron (MLP), k-Nearest Neighbors (kNN), and Random Forest (RF)—offers a reasonable mix of classifier types. MLPs represent neural networks, employing multiple perceptrons in conjunction to achieve the desired results. kNN is a non-parametric instance-based algorithm, which uses distance calculations between data points to make its predicitions. Finally, RF is an ensemble method, which trains multiple decision trees for its predictions. These choices provide a foundation for comparing various types of training methodologies, instance-based (k-NN) vs. model-based (MLP, RF) methods, for example.
- The implementation of both cross-validation and holdout set validation helps to ensure that the generated performance metrics are more reliable as indicators of a model's ability to generalize to new data. Hereby, cross-validation is used to mitigate the risk of overfitting to a single random train-test split.

2.1.3 Our strategy

As already mentioned, first step of our search for the best model training strategies consisted of trying out all possible combinations of datasets, preprocessing strategies and validation techniques. Accordingly, it looks like this:

```
COMBINATIONS = {
    "dataset": ["congress", "mushroom", "reviews", "seattle"], # chosen dataset
    "scaling": ["mean", "minmax", None], # preprocessing: scaling method
    "labeling": [True, False], # preprocessing: one-hot encoding or not
    "imputing": [True, False], # preprocessing: impute missing values or not
    "classifier_type": ["mlp", "knn", "rf"], # classifier type
    "cross_val": [True, False], # validation: cross-validation or holdout validation
}
random_combinations = list(itertools.product(*COMBINATIONS.values()))
random.shuffle(random_combinations)
```

Having defined our search space of 288 unique combinations, we proceeded to evaluate each configuration on the datasets. Since this process is computationally expensive, we decided to parallelize the grid search using the `multiprocessing` library. However, we encountered an issue with the `loky` backend in `sklearn`, which prevented us from running the grid search in parallel (fixed by using the hyperparameter `n_jobs=1`). Additionally for improved fault tolerance, we implemented a try-except block to catch exceptions, a custom signal based `@timeout((x))` decorator to limit the runtime of each configuration to `x` minutes, as well as a custom `@benchmark` decorator to measure the runtime of each configuration. Additionally we benchmarked the runtime of different disk I/O methods to avoid bottlenecks.

Here is an example of a single entry in the JSON file used to store the benchmark results:


```
...
{
    "results": [
        ...
        {
            "args": {
                "dataset": "congress",
                "scaling": "minmax",
                "labeling": "True",
                "imputing": "True",
                "classifier_type": "mlp",
                "cross_val": "True"
            },
            "classifier_config": {
                "hidden_layer_sizes": ["9", "6"],
                "activation": "tanh",
                "alpha": 7.82258e-5,
                "batch_size": "auto",
                "solver": "lbfgs",
                "learning_rate": "adaptive",
                "learning_rate_init": 0.002,
                "power_t": 0.7810684969,
                ...
                "beta_2": 0.998910637,
                "epsilon": 1.07e-8,
                "n_iter_no_change": "10",
                "max_fun": "15000"
            },
            "validation": "3-fold cross validation",
            "time": 0.019725292,
            "accuracy_score": 0.9495412844,
            "balanced_accuracy_score": 0.8946958553,
            "precision_score": 0.9495014653,
            "recall_score": 0.9495412844,
            "zero-one-loss": 0.0504587156
        },
        ...
    ]
}
```

Configuration fields:

- `args`: The configuration used for the run.
- `classifier_config`: The hyperparameters used for the classifier, which were selected based on a normal or uniform distribution for each classifier. We set the default value provided by `sklearn` as the mean of the distribution and adapted the standard deviation based on our intuition and observations. Since we will explore the impact of hyperparameters in the next step, we did not focus on this aspect in the initial grid search.
- `validation`: The validation strategy used for the run. Either 3-fold cross or holdout validation. We chose a 3-fold crossvalidation due to computational and time reasons.

Metrics:

- `time`: The runtime of the configuration in seconds [3] (excluding the time spent on disk I/O)

  Although we captured this metric, we later decided to discard it from the analysis, as we were concerned about the reliability of the measurements due to the limitations of our hardware and the potential impact of other processes running on the machine. We did not have the resources to run the configurations multiple times in an isolated environment for a more accurate measurement.

- `accuracy_score`: [4]

  Formula: $\frac{TP+TN}{TP+FP+TN+FN}$

  Accuracy is the correctness of both true positive and true negatives, among all samples. By normalizing the accuracy, we get the fraction of correctly classified samples. Collecting this metric is essential as it can be argued to amount to the general "quality" of the model.

  The best performance is 1 with `normalize == True` and the worst is 0.

- `balanced_accuracy_score`: [5]

  Formula: $\frac{\frac{TP}{TP+FN}+\frac{TN}{TN+FP}}{2}$

  By adjusting the balanced accuracy, we get the average of recall obtained on each class adjusted by chance. When true, the result is adjusted for chance, so that random performance would score 0, while keeping perfect performance at a score of 1. This metric is particularly useful for imbalanced datasets as it is the average of "true positive rate" and "true negative rate".

  When `adjusted=False` the best value is 1 and the worst value is 0.

- `precision_score`: [6]

  Formula: $\frac{TP}{TP+FP}$

  Precision is also called "positive predictive value" and is the likelihood of positive predictions to actually be correct. We decided to collect this metric because it could give some further insight as to why the accuracy might be low.

  The best value is 1 and the worst value is 0.

[3] The runtime was measured using the `time` module in Python. We used a Apple M2 Pro 14.2.1 23C71 with an arm64 CPU, Mac14,10 kernel version 23.2.0 and 16GB of RAM.

[4] As: `sklearn.metrics.accuracy_score(y, y_pred, normalize=True)`

[5] As: `sklearn.metrics.balanced_accuracy_score(y, y_pred, adjusted=True)`

[6] As: `sklearn.metrics.precision_score(y, y_pred, average="weighted", zero_division=0, labels=np.unique(y_pred))`

- `recall_score`: [7]

  Formlula: $\frac{TP}{TP+FN}$

  Recall is also known as "sensitivity" or "true positive rate" and can be viewed as the completeness of detected true positives, probability of detection or hit rate. We collected this metric with a similar reasoning as the one for the `precision_score`.

  The best value is 1 and the worst value is 0.

- `zero-one-loss`: [8]

  Formula: $\frac{FP+FN}{TP+FP+TN+FN}$

  The zero-one loss is the fraction of misclassifications, which is the sum of false positives and false negatives divided by the total number of samples. By normalizing the zero-one loss, we get the fraction of misclassified samples - otherwise it would be the number of misclassified samples.

  Since in our case `zero-one-loss` is the complement of `accuracy_score`, we decided to discard it from the analysis, as it doesn't provide any additional insights, although we captured it for completeness.

  The best value is 0 and the worst value is 1.

These metrics provide a well-rounded view of the model's performance (both the effectivity and the efficiency), across different datasets and configurations - effectively benchmarking the classifiers - and allow us to compare them for the next steps.

We believe that *these metrics are sufficient* to gain insights into the performance of the classifiers. Not only do they capture integral information for the analysis of the model's quality but also complementary / conflicting aspects of the model's performance. One such example is the trade-off between precision and recall [9].

2.2. Interpreting the results

We managed to cover 96.88% of our search space (9 of 288 runs failed).

To analyze the benchmark results, we first aggregated (`np.mean`) the performance metrics for each configuration across all datasets. This allowed us to identify the best-performing configurations. We found the best suited visualization for this task to be heatmaps. Before diving into the discussion of our learnings, which is based on the following four heatmaps that contain our chosen metrics, however, we also want to add a short evaluation of our results on the `time` metric. Intuitively, this evaluation does not have anything to do with the performance and quality of our models and is rather a quick statement about our experimental observations. The according heatmap is the last one of the five.

Most of the results on the `time` heatmap are not exactly noteworthy, but there are some combinations of classifier and dataset where training either takes a long time or times out completely. In general, the MLP classifier tends to take longer than its counterparts, especially on large and/or complex datasets. This makes sense given the sheer volume of calculations that have to be carried out in the training of an MLP model. This effect can be observed especially well for the `reviews` and `seattle` datasets which are both large and complex. The other notable observation is that the kNN classifier tended to time out often

[7] As: `sklearn.metrics.recall_score(y, y_pred, average="weighted", zero_division=0, labels=np.unique(y_pred))`

[8] As: `sklearn.metrics.zero_one_loss(y, y_pred, normalize=True)`

[9] Powers, David M W (2011). "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation". Journal of Machine Learning Technologies. 2 (1): 37–63.

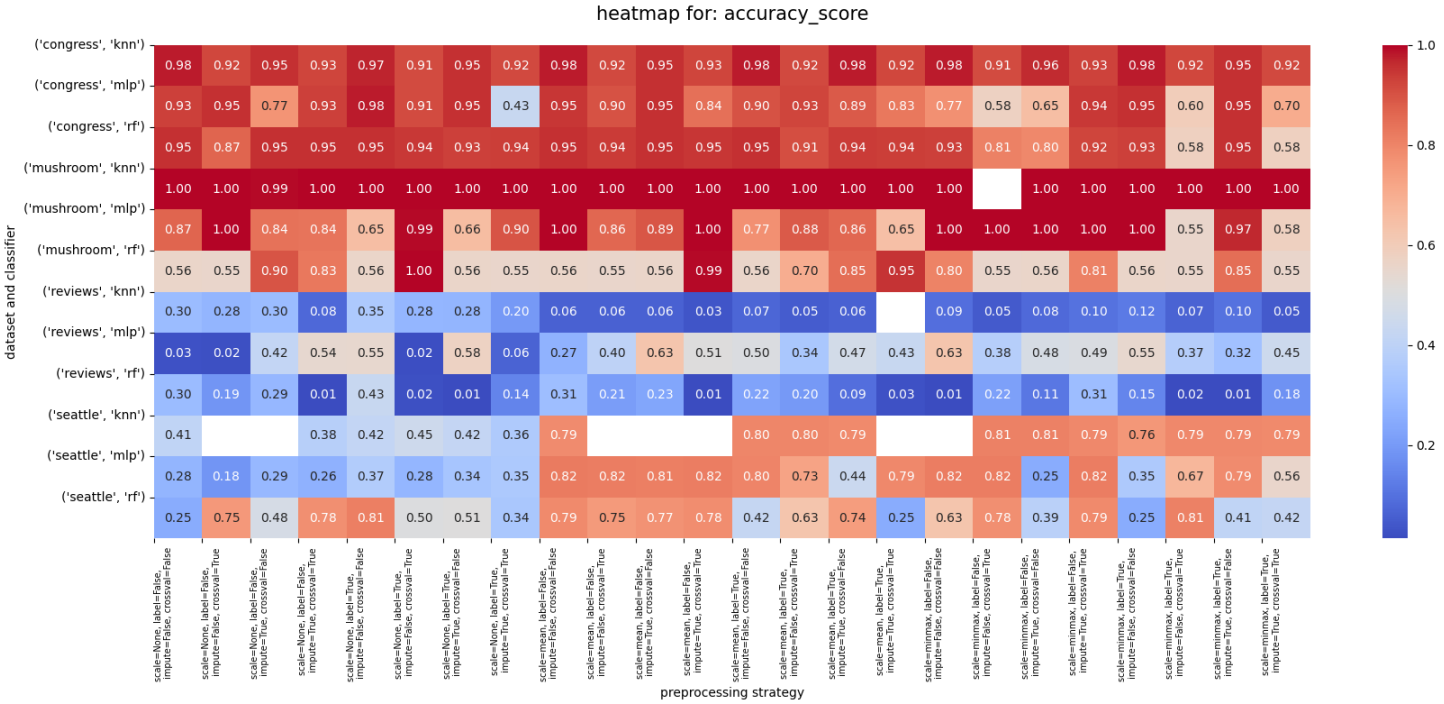

Figure 1: accuracy score heatmap

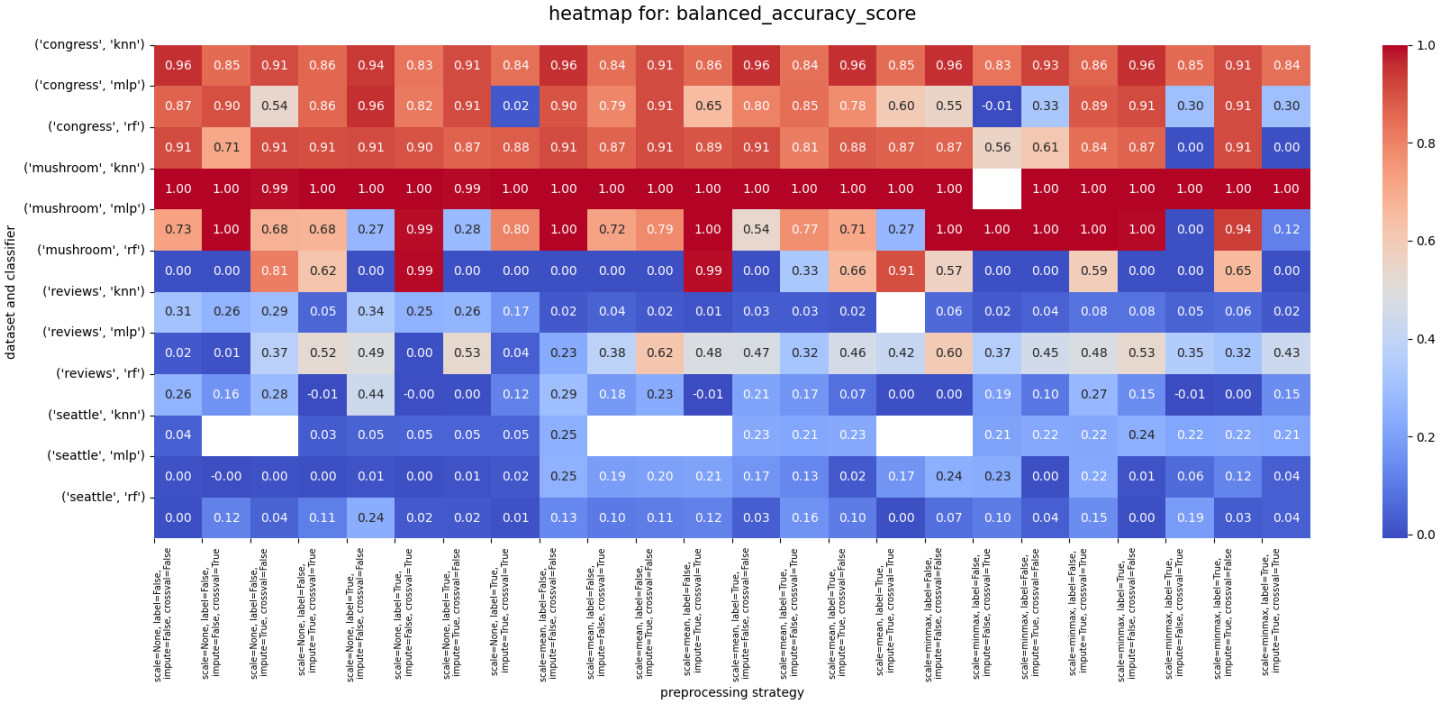

Figure 2: balanced accuracy score heatmap

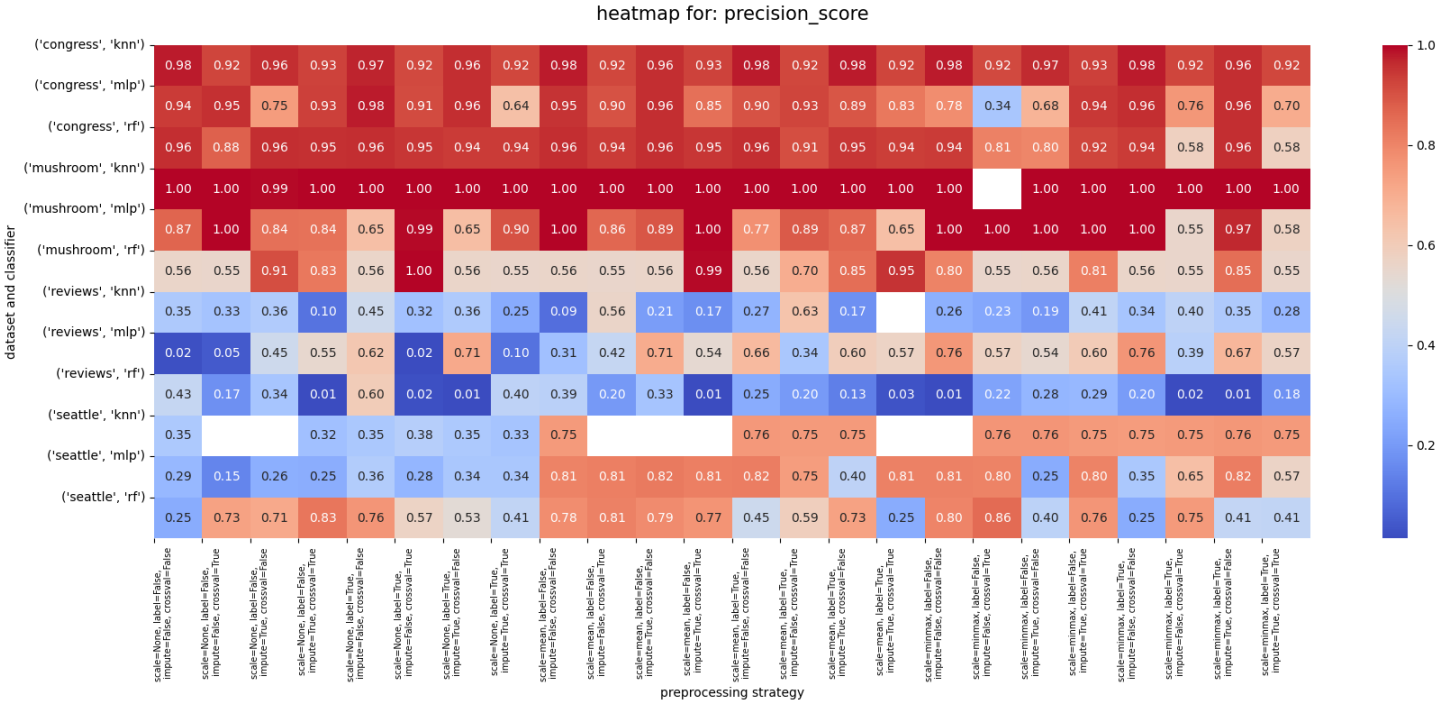

Figure 3: precision score heatmap

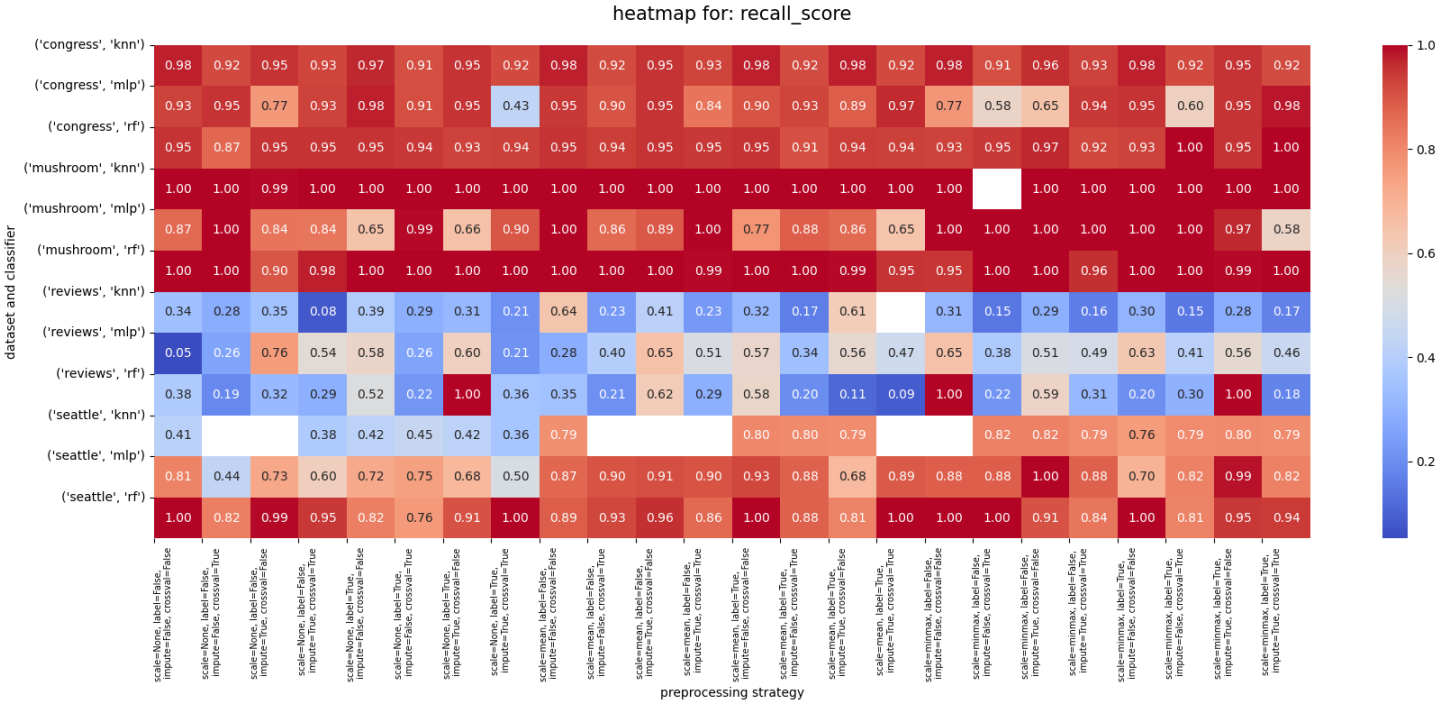

Figure 4: recall score heatmap

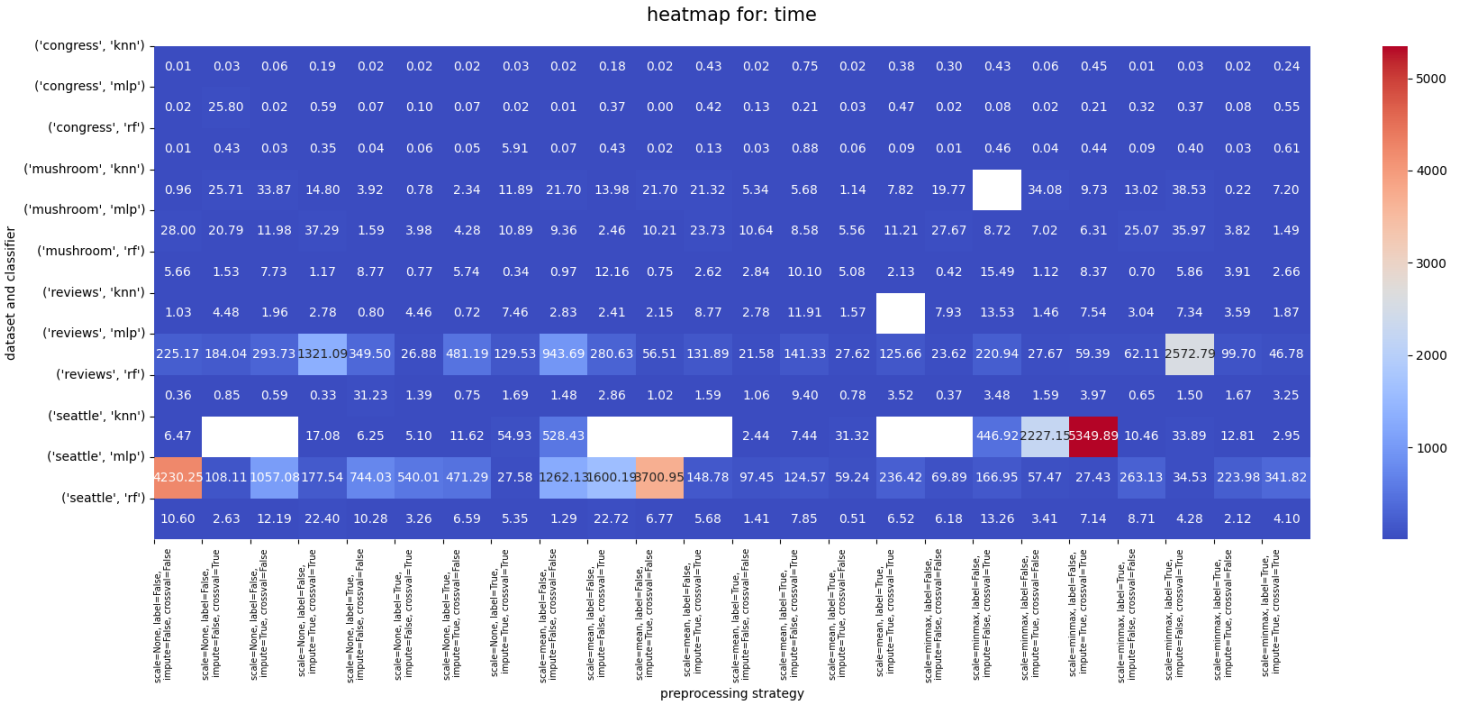

Figure 5: time heatmap

on the `seattle` dataset. We think that this could've been the case due to the fact that this dataset has a lot of different values for the class attribute, causing the prediction part of the training, which already takes a bit longer for kNN models, to sometimes skyrocket in terms of time taken.

2.2.1 Learnings

- **Congress Dataset**:

  The congress dataset was the easiest to work with, as all classifiers performed pretty well with respect to all metrics on this dataset, but the kNN classifier had the best max. and average score across all metrics and is accordingly the best choice for this dataset. The RF classifier also performed well on this dataset, but not as well as the kNN classifier. What mainly drags down the performance of the MLP and RF classifiers is the fact that there are some preprocessing combinations that led to drastically worse results than kNN achieved with those some configurations. Additionally, whenever kNN is actually outperformed by the other two classifiers, it is only by a marginal amount, which is why MLP and RF have trouble competing for kNN on this dataset.

  Out of all the preprocessing configurations, the best for the congress dataset with the **kNN classifier** across the board was `impute: False, cross_val: False`. The other two preprocessing parameters didn't influence the results as much.

- **Mushroom Dataset**:

  The kNN classifier had an almost *perfect* score on all metrics for the mushroom dataset indifferent to all other parameters and configurations. Thus, it should be the first choice for this dataset based on our initial results.

  The MLP and RF classifiers also had a perfect score with some configurations, but not as consistently as the kNN classifier.

Since the MLP classifier had more perfect scores than the RF classifier across all metrics and configurations and was more consistent in its performance, it should be considered the second choice for this dataset.

The best preprocessing configuration for the mushroom dataset with the **MLP classifier** by a wide margin was `scaling: minmax, label: False`. The other two preprocessing parameters did not have a significant impact on the performance as long as the mentioned configuration was used.

- **Reviews Dataset**:

  The review dataset was in general one of the most challenging to work with. Overall, all classifiers performed poorly on this dataset, with the MLP classifier showing the best performance by a wide margin. This could be due to the high dimensionality of the dataset, which makes it difficult for the kNN and RF classifiers to find patterns in the data. The imbalanced nature of the dataset could also be a factor, as the classifiers may struggle to learn from the minority classes.

  The best two preprocessing configurations for the reviews dataset with the **MLP classifier** were `label: False, impute: True, cross_val: False` with `mean` scaling and `label: False, impute: False, cross_val: False` with `minmax` scaling. There were a few other configurations with an accuracy above 0.5 but most were below this threshold.

- **Seattle Dataset**:

  The kNN timed out on far too many configurations to be considered a viable option for this dataset, so we won't consider it further.

  When looking at the MLP and RF classifiers, they showed a contrast in recall and precision, with the RF classifier having a higher average recall and the MLP classifier having a higher average precision. This suggests that the RF classifier is better at detecting true positives, while the MLP classifier is better at avoiding false positives. But their max. values were identical so it mostly depends on both the hyperparameters, processing and the given use case.

  In terms of accuracy however, the MLP classifier performed distinctly better than the RF classifier, which suggests that it is better at correctly classifying the samples overall.

  The best preprocessing configuration for the seattle dataset with the **MLP classifier** was `scale: mean, label: False`. The other two preprocessing parameters did not influence the results as strongly when this configuration was used. However, many other configurations showed some promise, too. As long as scaling was used, no matter if `mean` or `minmax`, the models were generally performing reasonably well, more often than not.

2.2.2 Validation

The effect of 3-fold cross validation vs. 80/20 holdout validation on the performance of the classifiers was also analyzed on a per-dataset basis. The results showed that it only had a significant impact on the congress dataset where there was a clear preference for the holdout validation strategy. This could be due to the small size of the dataset, which makes it easier to overfit to the training data when using cross-validation. The other datasets showed no clear preference for either validation strategy. This is the case for all of our chosen performance measures.

Excursion: Comparability

The comparisons between configurations above can be made because the metrics we chose for our analysis are independent of any outside factors and just strictly evaluate the performance of the model. One

metric we initially collected but then excluded from our analysis that is very dependent on outside factors is the time it took to train the model, which is also the reason why we excluded it.

The only thing that could make comparisons between our chosen metrics difficult is the fact that the training algorithms are not deterministic and can yield models of varying quality upon each execution, all things being equal. To combat this problem as best we can, we defined a random state that we used as a state parameter in all applicable models to keep the randomness to a minimum.

2.3 Fine-tuning hyperparameters

After finishing the systematic search for the most promising configurations, we continued onto the next step of our efforts, which was to focus on the best-performing configurations for each dataset and to find the best hyperparameters for it.

To achieve that goal, we employed multiple strategies. The most systematic one was our usage of the `GridSearchCV` class from `scikit-learn` to perform an exhaustive search over a specified parameter grid for each classifier. The results from this could be used as a good jumping-off point to perfect our parameter selection even more. We continued looking for better settings by essentially conducting some "manual hill-climbing" and following trends of increasing accuracies by accordingly changing the values of the hyperparameters. In this strategy, we mainly focused on the first couple of parameters, since experimentation quickly showed that these seemed to make the biggest difference concerning model quality.

Finally, to account for the case that we were too hyperfixated on our chosen configurations and hyperparameter trends and overlooked some "diamonds in the rough", we also trained some semi-randomized models (excluding configurations/hyperparameter values that have thus far only shown very bad results).

3. What did we learn from our experimentation?

First of all, although we have already discussed our results and learnings from our configuration exploration in 2.2.1, we would also like to showcase the details and actual numeric results of our best models, together with the preprocessing strategies and most important/sensitive hyperparameter settings used in their training.

For the `congress` dataset, we actually had a tie between kNN and RF concerning model performance, but we decided to present our results for RF here, since the settings for our final model are a little more interesting as we had to do more parameter testing to find the optimal setup. Additionally, this way, we can show our chosen parameters for all three classifiers we selected.

- `congress`:
  - classifier: RF
  - preprocessing strategies:
    - scaling: normal
    - encoding: one-hot
    - impute: mode
  - important hyperparameter settings:
    - n_estimators: 30
    - criterion: log_loss
    - max_depth: 50
    - min_samples_split: 2
  - metrics:
    - accuracy: 0.97727

        * balanced_accuracy: 0.96153
        * precision: 0.97846
        * recall: 0.97727
- `mushrooms`:
    - classifier: kNN
    - preprocessing strategies:
        * scaling: minmax
        * encoding: labeling
        * impute: mode
    - important hyperparameter settings:
        * n_neighbors: 5
        * weights: distance
        * algorithm: ball_tree
        * leaf_size: 1
        * p: 2
    - metrics:
        * accuracy: 0.99926
        * balanced_accuracy: 0.99856
        * precision: 0.99926
        * recall: 0.99926
- `reviews`:
    - classifier: MLP
    - preprocessing strategies:
        * scaling: normal
        * encoding: one-hot
        * impute: mode
    - important hyperparameter settings:
        * hidden_layer_sizes: [500]
        * activation: logistic
        * learning_rate: constant
        * learning_rate_init: 0.001
        * max_iter: 500
    - metrics:
        * accuracy: 0.69333
        * balanced_accuracy: 0.71099
        * precision: 0.76454
        * recall: 0.72222
- `seattle`:
    - classifier: MLP
    - preprocessing strategies:
        * scaling: normal
        * encoding: one-hot
        * impute: mode
    - important hyperparameter settings:
        * hidden_layer_sizes: [100]
        * activation: logistic
        * learning_rate: constant
        * learning_rate_init: 0.001
        * max_iter: 100
    - metrics:

* accuracy: 0.81732
* balanced_accuracy: 0.24673
* precision: 0.79985
* recall: 0.85731

There are quite a few takeaways from our experimentation with the diverse datasets at our disposal. First and foremost, as already mentioned in other parts of the report, model performance varies widely depending on dataset and configuration. Some of our datasets have one undisputed best classifier, independent of specific configuration parameters, others show promising results on multiple classifiers, depending on the specific configurations. This clearly reinforces the fact that there is no "one size fits all" solution when it comes to machine learning and that the dataset at hand always needs to be considered when deciding upon a model training strategy.

Another observation that confirms what we heard in lecture is that sometimes one singular decision - which might not even seem that impactful initially - can drastically change the quality of the resulting model. This especially applies to the preprocessing parameters. One such example is the comparison between the accuracy value of the MLP model for the `congress` dataset. In this case, the configuration `scale: none, label: True, impute: True, cross_val: True` produces a modul that is considerably worse than that of the configuration `scale: none, label: False, impute: True, cross_val: True`. As can be seen in the parameters, the difference between these two lies in the handling of categorical values, where the first one uses labelling and the second one uses one-hot encoding. To the untrained eye, it would seem like those two should not cause too great a difference in the model performance - both just encode the same categorical values after all. However, as our experimentation showed, simply the way in which the data is encoded can already make an immense difference on the model performance of some classifiers.

To sum up our findings, there is no clear winning classifier for all our datasets as their performance varies greatly between sets. Also the chosen preprocessing startegies make a considerable difference. While there are combinations of classifiers and preprocessing techniques and certain datasets, where only minor differences can be noticed (e.g. `congress` with kNN), this is an uncommon case. Finally, as is also obvious from the heatmaps, there are immense differences across the individual datasets. Any doubts about the actual diversity of our selection of datasets was cast away after analysing the results of our experimentation. When conversing about trying to find an optimal model, we always had to specify which dataset we were reffering to, since our best model training settings really were that different for each one.

To conclude, we are very content with our results, especially those for the `congress` and the `mushroom` datasets, as we achieved very good results in all metrics on those. However, also our performance on the other two sets is very satisfactory in our minds, especially given our resources. `seattle` was a somewhat difficult dataset to handle, yet we achieved an accuracy of above 80%, which exceeded our expectations. This is even more true for `reviews`, which was both a cryptic and quite difficult dataset to handle, both due to its dimensionality and numeric parameters, the meaning of which was completely unknown to us due to the undescriptive column namings. Through these barriers, we managed to achieve an accuracy of almost 70% which we are especially proud of.

We aren't aware of all too many systematic ways, in which our results could be drastically improved, as we already utilized all the techniques at our disposal to try and increase model performance as high as we could. Accordingly, our ideas concerning improvement either consist of following our current approach to its conclusion or restarting entirely. More specifically, one might try to follow the manual hill-climbing approach and choosing hyperparameter values accordingly. Similarly, one could also employ our final strategy of generating semi-random models in the hopes of finding something noteworthy. On the other hand, one could try to run the initial grid search again, hoping that the newly chosen values from the random distribution might align in such a way that a configuration that has been neglected thus far

shows promising results. This configuration could then again be further explored by employing the aforementioned strategies.

Aside:

As to aid the readability of our code, we decided to include a short explanation outlining the most important parts of our project structure. First of all, the `data` directory contains our datasets. We used the `reports` directory to store the most promising results from our experimentation, i.e., our best model training configurations. `main.ipynb` is the file that contains everything needed to train an individual model, including everything from setting preprocessing parameters, running the preprocessing and setting hyperparameter values to of course doing the actual training. The other scripts (except for `fetch_benchmark.py`) are used for our automatic combination testing strategy mentioned in the report.