Machine Learning

Exercise 1: Classification

Aleksandar Dadic, Paula Hilbert and Maximilian Lackner

November 20, 2022

1 Datasets

For this classification exercise we had to use 4 different datasets. In the following we want to give a quick overview of these and discuss their most important properties.

The first dataset is called "congressional voting". It has 16 attributes plus the target attribute 'class' and 218 instances. The target attribute can take one of the two values *democrat* and *republican*. The values are distributed in the following way:

democrat: 62% republican: 38%.

All the other attributes describe an important issue that the congressman of the U.S. House of Representatives voted on. Each instance of the dataset thereby represents the votes of one congressman on these 16 issues. The values of the attributes can be either y if the congressman voted for the issue, n if the congressman voted against the issue and *unknown* if the position is unknown. Therefore the goal of the classification is to predict the political party membership of a congressman based on their voting habits. All attributes are categorical and can only take two values. Because some of the values are missing i.e. are set to *unknown*, we will have to address data imputation in the preprocessing step. Also it is clear that we will not need to scale the data, because every attribute is binary.

The second dataset is called "Amazon Commerce Reviews". It has 10000 attributes plus the target attribute 'Class' and 750 instances, which represent reviews. The data is derived from customer reviews on Amazon. The target attribute has 50 different values. Each value is the first name of a different author. All other attributes represent an English word and take integer values, which describe how often the word was used in a review. The goal is to identify the author of a review based on the words that were used. Because the maximum values for each attribute can differ quite a bit, it could be useful to scale them in the preprocessing step. Furthermore there are no missing values.

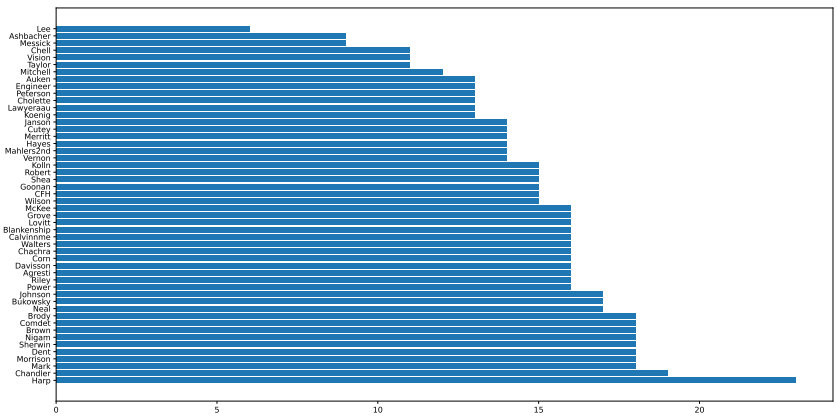


Figure 1: Distribution of the target attribute for "Amazon Commerce Reviews"

The "Census Income" dataset consists of data extracted from the 1994 Census database. The United States Census Bureau is a federal agency in the U.S. whose primary task is conducting the U.S. census every ten years. Using this data the goal is to predict whether a person earns more than 50K a year. The dataset has 14 attributes plus the target attribute 'income' and 48842 instances. The target attribute takes the values $>50K$ and $<=50K$ and is categorical. The values are distributed in the following way:

<= 50K: 76% > 50K: 24%.

The attributes 'age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss' and 'hours-per-week' are numerical, while 'workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex' and 'native-country' are categorical. Because the numerical values have rather different ranges we will probably need to scale them in the preprocessing step. Also the columns 'workclass', 'occupation' and 'native-country' have missing values, which must be addressed in this step too.

The last dataset is called "diabetes". This dataset contains medical data of females of Pima Indian heritage, which are at least 21 years old. All attributes are in some way linked to diabetes. Thus the goal of the machine learning model is to predict whether a person has diabetes or not. The dataset has 8 numerical attributes plus the nominal target attribute 'class' and 768 instances. The target attribute is categorical and takes the values *tested_negative* and *tested_positive*. The values are distributed in the following way:

tested_negative: 65% tested_positive: 35%

Because the value ranges of the numerical attributes differ a lot, we might need to scale them in the preprocessing step.

2 Classifiers

We now want to give an overview of the classifiers we used and argue why we choose them. We decided to use the random forest classifier, the linear support vector classifier and the ridge classifier from the scikit-learn Python package.

We definitely wanted to use at least one of the classifiers from the lecture, because we find it interesting to see how our theoretical knowledge can be applied to a practical problem. After looking at some examples on how other people solve classification problems, it was clear that random forests are frequently used. Furthermore we immediately got good results using this classifier on our first dataset "congressional voting". Thus we decided to work with this classifier. Also it is inherently multi-class, which is useful for the "Amazon Commerce Reviews" dataset. The random forest classifier belongs to the ensemble methods, which combine predictions of several base estimators to decrease variance. In this case the algorithm builds several decision trees and averages their predictions. This doesn't only lower variance but also reduces overfitting, which decision trees generally tend to do. On the downside bias can sometimes be slightly increased.

It was clear that finding a suitable classifier for the "Amazon Commerce Reviews" dataset was going to be a bit challenging because the dataset is very high dimensional. Since most of the values are 0 we are additionally dealing with sparse data. On top of that the dataset doesn't have a lot of instances, which means that there is very limited information available for each of the 50 classes. After some research we found out that support vector machines are effective for high dimensional sparse data. These algorithms are designed to solve binary classification problems, but can also be used for multi-class datasets using a compatible strategy. A support vector machine constructs a hyperplane or a set of hyperplanes to separate the data points into two classes. The hyperplanes are chosen to maximize the distance to the nearest data points of both classes. Sometimes this will not be effective, so one can use a kernel to map the data points to a higher-dimensional space, where the separation is easier. In general, there are two support vector machine classifiers in scikit-learn, namely the linear support vector classifier and the support vector classifier, of which the first one is recommended for large datasets. One can set the kernel to linear in the second classifier, but it will not yield the same results as the linear support vector classifier, because the underlying library is different. Testing them both on our dataset, we got the best results with the linear support vector classifier, which is why we chose this as our second classifier. For multi-class data it uses the "one-vs-the-rest" strategy, therefore training as many models as there are classes for the target attribute.

For the last classifier we wanted to choose something simple that is still quite effective. After looking through the different classifiers scikit-learn has to offer, we ended up focusing on linear models. We did some simple tests with our datasets and were quite impressed with the effectiveness and efficiency of the ridge classifier. Therefore we decided to use this classifier for our project. Ridge regression is an adaptation of linear regression, where a penalty term is added to the cost function. This term is introduced to prevent overfitting. The penalty term itself contains a hyperparameter which controls the

amount of regularization. If this hyperparameter is set too large, then underfitting can occur. This regression method can be used for binary classification, when the target attribute is transformed to the values $\{-1, 1\}$. The ridge classifier employs multi-output regression for multi-class classification. In multi-output regression two or more numerical outputs are predicted for each input.

3 Evaluation

There are a few things to consider when evaluating the performance of a classifier on a given dataset. We had to decide which performance measures we want to use as well as how to split our data into training and test sets. For each classifier and dataset we used repeated stratified k-fold cross-validation. Here the available data is divided into k groups of samples (called folds) of the same size. In each step $k-1$ groups are used for training and the remaining one for validation, hence the classifier is trained k times. The descriptor 'stratified' means that each of those folds are chosen in a way to retain approximately the same distribution of classes for the target attribute of the whole dataset, whereas 'repeated' means that the data is repeatedly split like this for a certain amount of times. The advantages of validating a classifier in this way are plentiful. Firstly, using multiple folds reduces the huge impact one split has on the results by not just dividing the data into two sets, but rather computing the performance for k different splits and averaging the results. Repeating this process further decreases variance of the results. Using stratified cross-validation is especially useful when dealing with imbalanced data because it ensures that the distribution of classes is properly represented in every training/test set. However, deciding to evaluate a model like this leads to other challenges concerning data leakage. What this exactly entails and how we dealt with it will be discussed in the preprocessing section.

Before moving on to the performance measures, we want to briefly discuss what must be considered when choosing the number of folds and repetitions. With a higher number of repetitions the results will generally reflect the performance of a classifier more accurately, but at some point increasing the repetitions will hardly make a difference. Since the runtime of the evaluation also grows with each repetition, a compromise must be made. For the number of folds one must also consider the number of instances and the sizes of each class: If a dataset is rather small a too large number of folds can lead to too small test sets. Also if one class has only a small number of instances, then too many folds will lead to some not containing a sample of this class. As an alternative to cross-validation one could use the holdout method, where the data is only split once. However, this is not a recommended validation technique except for datasets with a high number of instances (we will verify this in the "Results and Findings" section).

We now want to focus on performance measures. Here we can differentiate between effectiveness and efficiency. Starting with efficiency, this describes the time it takes to train the classifier and the time it takes to classify new data with the model after training. Which time is more important depends on the use case: in most real-life applications you want a fast classification, whereas the training duration might be less interesting because training is only done once. The goal of this project was to compare different preprocessing steps as well as different hyperparameters for three classifier and find the best performing settings. Since this means we had to evaluate each classifier very often, we had to compute lots of repeated stratifies k-fold cross-validations, which always includes both training and testing. So concerning efficiency both the time for training and testing were important to us, although generally the time it takes to train a model is much longer than classifying new data. We measured efficiency by how long it took to train and test the classifier on one split (i.e. holdout method), where 70% of data was used for training and the other 30% for testing. We used this measure because depending on the dataset and classifier the number of folds and repetitions varied, so we wanted to compare the runtime for the classifier fairly.

For measuring effectiveness there are some more possibilities. The most widely used metric is accuracy, which is calculated by dividing the number of correct predictions by the total number of predictions. It gives a pretty good idea on the performance of the classifier but can be misleading if the dataset is imbalanced. For example if 99% of samples belong to one class and the other 1% to a second class, then even the dummy classifier (i.e. labeling everything with the majority class) would have 0.99 accuracy. Some could say this is a great result, but the high accuracy is deceptive in this example, because the classifier actually didn't learn anything. The "Amazon Commerce Reviews" and the "Census Income" datasets are imbalanced but not to a terrible degree, so we decided to use accuracy as a metric for each dataset. However, we kept in mind that it might not be the best measure for these two datasets. Two other popular metrics are precision and recall. Precision describes what proportion of all positive predictions were correct and recall is given by dividing the number of true positives by the sum of true

positives and false negatives. Thus high precision minimizes false positives and high recall minimizes false negatives. A model is considered to be better the higher these values are, but for most models they are inversely related. So instead of having to choose one of them, the so-called F-scores combine them. For all datasets except for "diabetes" we choose to use the F1 score, which is defined as

$$F1 = \frac{2 \text{ precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

The closer the F1 score is to 1, the better the model. On the downside, a low F1 score is not really informative, because it could be both due to precision or recall. An advantage of recall, precision and F1 score is that they are suitable for imbalanced data. This is why we chose to use the F1 score in 3 out of 4 datasets; only in the "diabetes" dataset we chose to use recall instead, because the main goal in that case should be to not falsely diagnose a sick person as healthy. Moreover we don't mind having false positives as people usually will get more medical testing done if the model classifies them as *tested_positive*. In table 1 we summarised which effectiveness measure we chose for each dataset.

Dataset	measure 1	measure 2
congressional voting	accuracy	F1 score
Amazon Commerce Reviews	accuracy	F1 score (macro)
Census Income	accuracy	F1 score
diabetes	accuracy	recall

Table 1: Used effectiveness measures

In the short description above on the F1 score we only discussed how binary data is handled, but the "Amazon Commerce Reviews" dataset is multi-class. In scikit-learn there are three possible ways to calculate the F1 score for multi-class data, which are selected by setting the parameter `average` either to 'micro', 'macro' or 'weighted'. We decided to use 'macro', where the F1 score is calculated for each class in a "one-vs-the-rest" manner and then averaged. Using this strategy the F1 score of every class is counted equally, in comparison to 'weighted' where the F1 score of each class is weighted in the same proportion as that class is present in the data.

However, even though a lot of thought went into the selection of our performance metrics, they are not "sufficient" in the sense that we can't guarantee with 100% certainty that the trained model will perform as well on unseen data. We can only try to select the best metrics according to the given data and the use case of the model. But ultimately there is no metric that can predict the future.

4 Preprocessing

We consider four different preprocessing tasks: data imputation, encoding of categorical data, scaling of numerical data and dimension reduction. In this section we want to have a look at each dataset and describe which of these tasks need to be applied and what settings are available for each task. We will also compare these settings and find the best preprocessing strategy for each dataset and classifier by comparing the effectiveness measures described in the last section. In order to have a meaningful comparison we used the default hyperparameters for each classifier and ensured to split the data identically in the cross-validation for each pair of dataset and classifier.

We also took some measures to avoid data leakage. Data leakage occurs when the data is preprocessed before it was split into the training and test set. This way information from the test set might be used on the training set (for example for data imputation). To avoid this it is essential to always preprocess the training data first and then afterwards apply the same transformations on the test set. For example: The MinMaxScaler uses the minimum and maximum value of each attribute for scaling. However, those values must only be calculated on the training set and then used on both the training and test set to avoid data leakage. This method seems difficult when using cross-validation because of the many splits. However, scikit-learn has an easy solution for this, namely pipelines, which guarantee that the right order of procedures is kept for each split. We therefore used pipelines in every preprocessing task.

Congressional voting: Firstly we want to note that the missing values are labeled as *unknown* in this dataset. Because all attributes are categorical there are mainly two strategies for data imputation: we can either replace all missing values with the most frequent value of the respective attribute or we can treat the value *unknown* as a separate category for the attribute. The categorical data was encoded

using one-hot-encoding except for binary attributes, which were just relabeled with 0 and 1. So already depending on the data imputation the encoding will be different. But either way we will always end up with binary attributes after the encoding, which is why there is no need for scaling. Also dimension reduction is needless, as the dataset is already really small. So in the end we only compare the two possibilities for data imputation. Table 2 shows the results for each classifier, where the first score in each cell is accuracy and the second number is the F1 score.

Data Imputation	Random Forest	LinearSVC	Ridge
most frequent	0.972/0.963	0.966/0.957	0.968/0.959
unknown is category	0.967/0.958	0.965/0.954	0.967/0.957

Table 2: Comparison of data imputation for "congressonal voting" dataset

We can see that replacing missing values with the most frequent category of the respective attribute was always the better strategy according to both effectiveness measures. For this experiment we used 5 folds and 10 repetitions in the cross-validation.

Amazon Commerce Reviews: In this dataset there are no missing values, so we don't have to consider data imputation. The target attribute has 50 different classes which we encoded with the values $0, \ldots, 49$. Since all other attributes are numerical we will test different scaling methods. However, we will only consider scalers that don't destroy the sparsity of the data, because we had terrible results with other methods. Considering that there are two possible scaling options in scikit-learn: The MaxAbsScaler and the StandardScaler with the parameter setting `with_mean = False`. The MaxAbsScaler divides each feature by the maximum absolute value of the respective attribute, while the StandardScaler with the described setting divides each feature by the standard deviation of the attribute. We also tested one method for dimension reduction called principal component analysis (PCA) with `n_components` set to 0.95. Again we will present our findings in a table below, where the first score in each cell is accuracy and the second number is the F1 score (macro averaged).

Scaler	Dimension Reduction	Random Forest	LinearSVC	Ridge
MaxAbsScaler	None	0.572/0.523	0.730/0.696	0.708/0.675
StandardScaler	None	0.572/0.521	0.711/0.677	0.69/0.657
None	None	0.57/0.52	0.603/0.572	0.578/0.546
MaxAbsScaler	PCA	0.217/0.183	0.680/ 0.646	0.719/0.677
StandardScaler	PCA	0.239/0.206	0.569/0.505	0.712/0.673
None	PCA	0.249/0.209	0.47/0.44	0.608/0.564

Table 3: Comparison of preprocessing methods for "Amazon Commerce Reviews" dataset

In the above experiment we used 5 folds and 10 repetitions in the cross-validation. It is evident that the MaxAbsScaler performed best for all classifiers. However, only the ridge classifier benefited from the dimension reduction. We also tested different values for the parameter `n_components`, which selects the number of components of the PCA such that the amount of variance is greater than its value. There we achieved the best performance with `n_components` set to 0.99.

Census Income: This dataset has missing values, but only for categorical attributes. Those values are labeled with *?*. For the categorical attributes we used the same two strategies for data imputation and encoding as with the "congressional voting" dataset. The only exception to this was the 'education' column, which we deleted because it was already encoded in the 'education-num' attribute. For the numeric columns we tested different scalers. Because we don't have to consider sparsity for this dataset, there are more options in scikit-learn, namely StandardScaler, MinMaxScaler, MaxAbsScaler, RobustScaler, PowerTransformer and QuantileTransformer. We already know the first two scalers from the lecture and the third from the last dataset. The RobustScaler subtracts the median from each value and divides the result by the interquartile range of the respective attribute. This makes it robust to outliers. The PowerTransformer makes the values more Gaussian-like, while the QuantileTransformer transforms the data so that each attribute is uniformly distributed afterwards. This transformation is also robust to outliers. Since the dataset isn't too big, dimension reduction seemed unnecessary. Like for the previous datasets we will present the result in a table, where the first score in each cell is accuracy and the second number is the F1 score.

Scaler	Data Imputation	Random Forest	LinearSVC	Ridge
StandardScaler	most frequent	0.857/0.677	0.852/0.655	0.84/0.607
MinMaxScaler	most frequent	0.857/0.676	0.851/0.654	0.84/0.608
MaxAbsScaler	most frequent	0.857/0.676	0.852/0.654	0.84/0.608
RobustScaler	most frequent	0.857/0.676	0.85/0.648	0.779/0.203
PowerTransformer	most frequent	0.852/0.665	0.845/0.64	0.842/0.618
QuantileTransformer	most frequent	0.857/0.676	0.845/0.645	0.842/0.621
None	most frequent	0.857/0.676	0.8/0.378	0.779/0.236
StandardScaler	*?* is category	0.857/0.675	0.853/0.659	0.841/0.61
MinMaxScaler	*?* is category	0.857/0.675	0.853/0.658	0.841/0.61
MaxAbsScaler	*?* is category	0.857/0.676	0.853/0.658	0.841/0.61
RobustScaler	*?* is category	0.857/0.676	0.852/ 0.654	0.78/0.203
PowerTransformer	*?* is category	0.852/0.666	0.846/0.645	0.843/0.623
QuantileTransformer	*?* is category	0.857/0.675	0.847/0.649	0.844/0.627
None	*?* is category	0.857/0.675	0.8/0.379	0.78/0.236

Table 4: Comparison of preprocessing methods for "Census Income" dataset

We used 5 folds and 5 repetitions in the cross-validation for this experiment, except for the random forest classifier, where we lowered the repetitions to 1 in order to speed things up. For this dataset the best preprocessing strategy was different for every classifier. The respective cells are marked in green in the table above. After these experiments we decided to use the StandardScaler combined with setting missing values as the most frequent category for the random forest classifier, the StandardScaler but with setting missing values as an extra category for the linear support vector classifier and the QuantileTransformer combined with setting missing values as an extra category for the ridge classifier.

Diabetes: This dataset doesn't have missing values, only consists of numeric attributes (except for the target attribute) and is very low dimensional, which is why we only tested different scaling methods. We used the same scalers as in the last dataset except for the QuantileTransformer. Again we will present our findings in a table below, where the first score in each cell is accuracy and the second number is recall.

Scaler	Random Forest	LinearSVC	Ridge
StandardScaler	0.766/0.589	0.77/0.56	0.77/0.555
MinMaxScaler	0.767/0.592	0.77/0.552	0.768/0.542
MaxAbsScaler	0.767/0.59	0.771/0.553	0.769/0.543
RobustScaler	0.766/0.589	0.77/0.56	0.77/0.555
PowerTransformer	0.766/0.592	0.763/0.564	0.761/0.554
None	0.766/0.588	0.673/0.47	0.77/0.555

Table 5: Comparison of different scalers for "diabetes" dataset

We used 5 folds and 10 repetitions in the cross-validation for this experiment. In this case the two effectiveness measures gave us different scalers as the best choice. We marked the best scaler for accuracy in blue and the best scaler for recall in red. Our goal was to choose a scaler where both metrics are pretty high. Our choices are labeled in green. As easily extracted for the table we will use the MinMaxScaler for random forest and the StandardScaler for the linear support vector and the ridge classifier.

5 Hyperparameters

In this section we want to take a closer look at the hyperparameters of our three classifiers. We want to focus on two different aspects: the tuning of hyperparameters and a sensitivity analysis of those hyperparameters. First we will discuss the tuning of hyperparameters, where we want to identify all the important hyperparameters for each classifier and define appropriate parameter ranges. We do this because we want to use Bayesian Optimization from the scikit-optimize package for parameter tuning. This optimization method takes a search space as an input and iteratively tries to find the best model with respect to a given performance measure. This is not done completely randomly but with a clever strategy, where the optimization builds a probability model of the objective function to find its maximum. In our case the

objective function is equal to the performance measure. Since we settled on two effectiveness measures for each dataset, we will do the parameter tuning twice, i.e. once for each measure, and then compare the results.

Random Forest: This algorithm has quite a lot of parameters and is relatively slow compared to the other classifiers. This is why we decided to focus on the following five hyperparameters, which greatly influence the structure of the forest:

- `n_estimators`: Gives the number of trees in the forest.
- `max_depth`: Is the maximum depth of each tree. When set to 'None', the whole tree is constructed or it is expanded until all leaves contain less than `min_samples_split` instances.
- `min_samples_split`: Describes the minimum number of samples necessary to split an internal node.
- `min_samples_leafs`: Gives the minimum number of instances required to be at a leaf node. This means a split is only considered if it results in at least `min_samples_leafs` instances in both the left and right branches.
- `max_features`: Takes the values 'sqrt', 'log2' and 'None', where 'sqrt' is the default. It describes the number of attributes to examine when searching for the best split.

For the bigger datasets "Amazon Commerce Reviews" and "Census Income" we choose the following search space:

$$\texttt{n\_estimators} \in \{100, \dots, 2000\}, \quad \texttt{max\_depth} \in \{\text{None}, 10, \dots, 100\},$$
$$\texttt{min\_sample\_split} \in \{2, \dots, 10\}, \quad \texttt{min\_sample\_leaf} \in \{1, \dots, 5\},$$
$$\texttt{max\_features} \in \{\text{sqrt}, \text{log2}\}.$$

It is important to note that we didn't consider 'None' as a possible setting for the parameter `max_features`, because this resulted in a way too long runtime and it is empirically known, that the other settings give better results for classification. However, as the other two datasets are much smaller we could increase the number of trees to 3000 and also test `max_features`='None'. Otherwise the search space for Bayesian Optimization stayed the same. In table 6 we summarized the best parameter settings when using accuracy in the optimization.

Dataset	`estimators`	`depth`	`split`	`leafs`	`features`	accuracy
congressional voting	2007	100	2	1	log2	0.973
Amazon Reviews	2000	100	2	1	sqrt	0.721
Census Income	1448	40	17	2	sqrt	0.866
diabetes	2992	90	2	4	sqrt	0.773

Table 6: Tuned hyperparameters for random forest and accuracy

For all datasets except for "diabetes" we used the F1 score as a second metric. The parameters with highest accuracy always coincided with more or less the best F1 score we achieved during tuning. The F1 scores for the hyperparameters in table 6 were 0.966 for "congressional voting", 0.667 for "Amazon Commerce Reviews" and 0.684 for "Census Income", while the best scores were 0.966, 0.678 and 0.686. Therefore we concluded that the best hyperparameters are the ones in table 6 for the datasets "congressional voting", "Amazon Commerce Reviews" and "Census Income". Looking at the "diabetes" dataset the decision isn't quite as easy, because the best hyperparameters for recall are very different to the ones in the table above. For recall the tuned parameters are

$$\texttt{n\_estimators: } 3000, \ \texttt{max\_depth: } 50, \ \texttt{min\_sample\_split: } 6, \quad \texttt{min\_sample\_leaf: } 1, \ \texttt{max\_features: } \text{'None'} \tag{1}$$

with a recall of 0.623. Changing `max_depth` to 100 doesn't change the recall score but improves accuracy. Because recall is the more important measure for us in that dataset, we would want to use the parameters (1), but only if that doesn't influence accuracy too negatively. Computing accuracy for the parameters (1) gives us the result 0.766. We are fine with it being a bit lower as a trade-off for having the best recall.

Linear Support Vector Classifier: This classifier is most influenced by the regularization parameter `C`. However, we also considered two more parameters which seemed very interesting after some initial experiments. We again want to list these parameters and explain their meaning.

- `C`: Is called regularization parameter and the strength of regularization is inversely proportional to it. It has to be greater than 0.
- `fit_intercept`: Can be 'True' or 'False' and decides if the intercept is calculated for this model.
- `class_weight`: We considered the values 'balanced' or 'None' (default). Each class j has its own parameter C, which is set to `class_weight[j]`*C, where the weights are 1 for the setting 'None' and the weights are inversely proportional to the class frequency in the training data for the setting 'balanced'.

The influence of the `class_weight` parameter was especially interesting for our imbalanced datasets. For all four datasets we used the search space

$$\texttt{C} \in \left[10^{-6}, 10^{6}\right], \quad \texttt{fit\_intercept} \in \{\text{True}, \text{False}\}, \quad \texttt{class\_weight} \in \{\text{balanced}, \text{None}\}$$

for tuning. In table 7 we display the best parameter settings, when using accuracy as a measure for success.

Dataset	`C`	`fit_intercept`	`class_weight`	accuracy
congressional voting	0.355	True	None	0.967
Amazon Commerce Reviews	0.023	True	balanced	0.742
Census Income	0.153	False	None	0.853
diabetes	0.215	True	None	0.771

Table 7: Tuned hyperparameters for linear support vector classifier and accuracy

We also want to look at the best parameter settings, when using F1 score/recall (depending on the dataset) as the optimization target. This is shown in table 8.

Dataset	`C`	`fit_intercept`	`class_weight`	F1 score/recall
congressional voting	0.586	True	None	0.961
Amazon Commerce Reviews	0.024	True	balanced	0.71
Census Income	0.274	True	balanced	0.677
diabetes	0.004	False	balanced	0.795

Table 8: Tuned hyperparameters for linear support vector classifier and F1 score/recall

For the datasets "congressional voting" and "Amazon Commerce Reviews" the parameter values are the same for the parameters `fit_intercept` and `class_weight` in table 7 and 8, so obviously they are the best choice. However the parameter `C` is slightly different. We will later verify in the sensitivity analysis that such small changes to the parameter `C` hardly influence the performance measures. From these observations we derived that a good setting is `C` = 0.58 for "congressional voting" and `C` = 0.02344 for "Amazon Commerce Reviews". Finding the best parameters for both metrics simultaneously is more difficult for the datasets "Census Income" and "diabetes". Since the "Census Income" dataset is imbalanced, we want to prefer a higher F1 score, hence settling on the parameter setting shown in table 8. Using these parameters we achieve an accuracy of 0.805. On the other hand for the "diabetes" dataset we need further analysis of the influences of each parameter, which we will conduct later in the sensitivity analysis. Here the goal will be to find settings, so that both measures are as good as possible simultaneously. There we can use the scores from table 7 and 8 as reference, because they contain the highest scores we were able to achieve when maximizing the two measures separately.

Ridge Classifier: Here we also have one "main" hyperparameter (`alpha`), but we again also tuned the parameter `fit_intercept`. The hyperparameter `alpha` describes the regularization strength and corresponds to $\frac{1}{2\texttt{C}}$ in the linear support vector classifier. For all datasets we chose the search space

$$\texttt{alpha} \in [0, 100], \quad \texttt{fit\_intercept} \in \{\text{True}, \text{False}\}.$$

In table 9 below we summarized the best parameter settings when optimizing for accuracy.

Dataset	alpha	fit_intercept	accuracy
congressional voting	0.057	False	0.968
Amazon Commerce Reviews	51.566	False	0.743
Census Income	94.59	True	0.844
diabetes	6.137	True	0.771

Table 9: Tuned hyperparameters for ridge classifier and accuracy

We again want to compare these settings to the tuned parameters when using F1 score/recall as the optimization target. The tuned parameters are shown in table 10:

Dataset	alpha	fit_intercept	F1 score/recall
congressional voting	0	False	0.959
Amazon Commerce Reviews	33.391	False	0.707
Census Income	0	False	0.628
diabetes	39.982	False	0.795

Table 10: Tuned hyperparameters for ridge classifier and F1 score/recall

For the dataset "congressional voting", the parameter `fit_intercept` is the same in both tables and the values for `alpha` are very similar. Calculating the F1 score for `alpha` = 0.057 gives us 0.959. This is the same score as in table 10, so the best parameter setting is the one given in table 9. For the dataset "Amazon Commerce Reviews" we also calculated the F1 score for the parameters in table 9, which came out to be 0.706. This again is hardly different from 0.707. Thus the best hyperparameters are again those given in table 9. For the dataset "Census Income" we preferred a better F1 score, so we calculated the accuracy for the settings in table 10. There the result was 0.844, which is the same value as in table 9. So we concluded that the best hyperparameter settings are those shown in table 10. Finally, for the "diabetes" dataset it is not clear from the tables above which parameters are the best, so we will decide on the best setting after gathering additional information from the sensitivity analysis.

For every tuning above we used cross-validation with 5 folds and 10 repetition except for the datsets "Amazon Commerce Reviews" and "Census Income", because the runtime would have been too long otherwise. There we used 5 repetition instead, except for the random forest classifier, where we lowered the repetitions to 1. The same applies to the sensitivity analysis below.

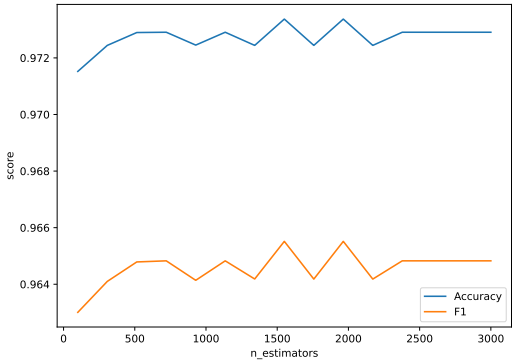
Now we want to conduct a sensitivity analysis of the hyperparameters, i.e. analyzing the effect of single parameter changes using the tuned parameters from above for each classifier and dataset pair. We will achieve this by plotting the effectiveness measures against each hyperparameter. The dataset "diabetes", however, will receive a slightly different treatment for the linear support vector classifier and the ridge classifier: There we will try all available combinations of the attributes `fit_intercept` and `class_weight` and vary the parameters `C` and `alpha` in order to conclude the hyperparameter tuning from before.

Random Forest: Our first goal is to investigate how much each of the 5 hyperparameters of the random forest classifier influences its effectiveness. We found out that this highly depends on the dataset:

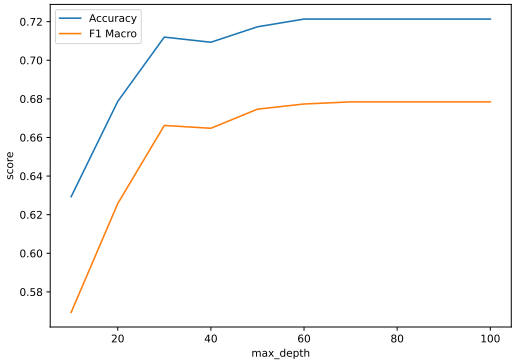
- `n_estimators`: For the datasets "Census Income" and "diabetes" varying this parameter hardly changed the effectiveness scores, while for the dataset "Amazon Commerce Reviews" the higher the value the better the result was. On the contrary, for the "congressional voting" dataset this parameters influenced accuracy and F1 score but with no clear trend.
- `max_depth`: This hyperparameter had no influence on the dataset "congressional voting", but for the other datasets the effectiveness increased with a higher value.
- `min_samples_split`: For the datasets "Census Income" and "diabetes" varying this parameter hardly changed the effectiveness scores. However, this parameter had a clear impact on other two datasets, where generally lower values fared better.
- `min_samples_leafs`: Here the best value either was one or two for all datasets and increasing the values always lowered effectiveness. This parameter had the least impact for the datasets "Census Income" and "diabetes".

- `max_features`: The best value for accuracy and F1 score was 'sqrt' across all datasets. However, the value 'None' produced the best recall score.

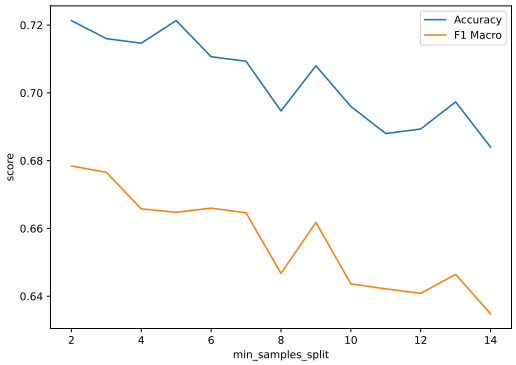
To underline these findings we show a few exemplary plots for different hyperparameters and datasets in figure 2 below.



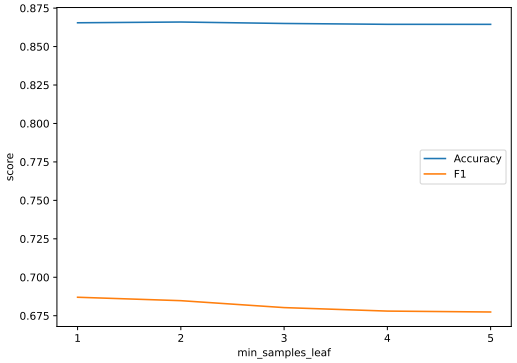
(a) Parameter `n_estimators` for "congressional voting" dataset



(b) Parameter `max_depth` for "Amazon Commerce Reviews" dataset



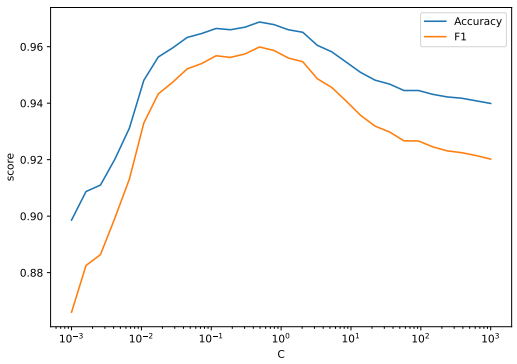
(c) Parameter `min_samples_split` for "Amazon Commerce Reviews" dataset



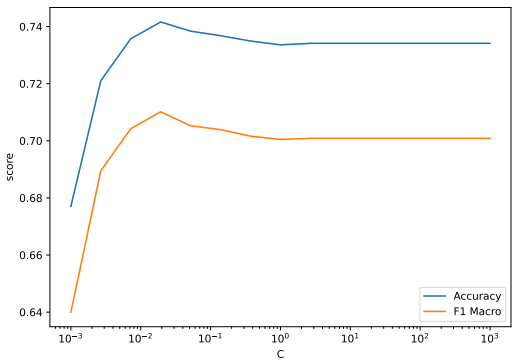
(d) Parameter `min_samples_leaf` for "Census Income" dataset

Figure 2: Sensitivities for Random Forest

Linear Support Vector Classifier: The hyperparameter `C` influences the performance significantly across all datasets. Depending on the data this influence can vary in magnitude. As an example we show the plots for the datasets "congressional voting" and "Amazon Commerce Reviews" in figure 3.



(a) "congressional voting"



(b) "Amazon Commerce Reviews"

Figure 3: Sensitivity of the parameter `C`

Analysing our plots we can also establish, that the parameter `class_weight` definitely influences all three performance measures accuracy, F1 score and recall. However, it is not possible to say which value will

be the preferred one, because it depends on the dataset and the performance measure. In contrast setting `fit_intercept` to 'True' always improved accuracy and F1 score, but 'False' was the better choice for recall for the dataset "diabetes".

We still have to analyze the plots for the "diabetes" dataset to conclude parameter tuning. Looking at the plots in figure 4 we find that the best parameter setting is `alpha` = 0.028, `fit_intercept` = 'False' and `class_weight` = 'balanced' when favoring recall over accuracy. These parameters give us an accuracy of 0.736 and a recall of 0.796.

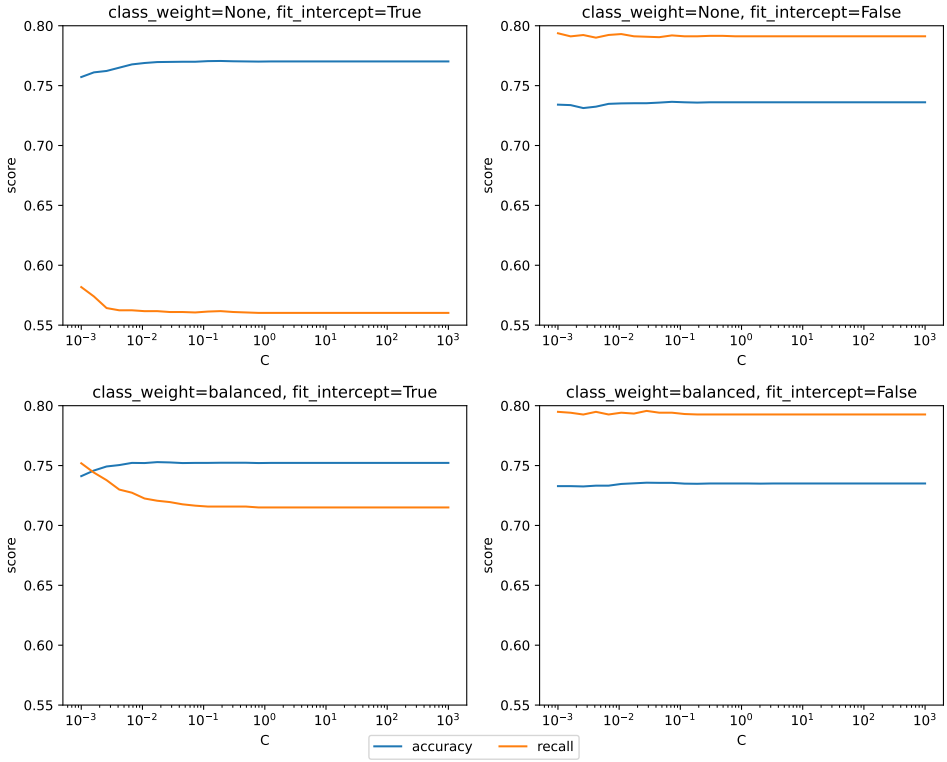
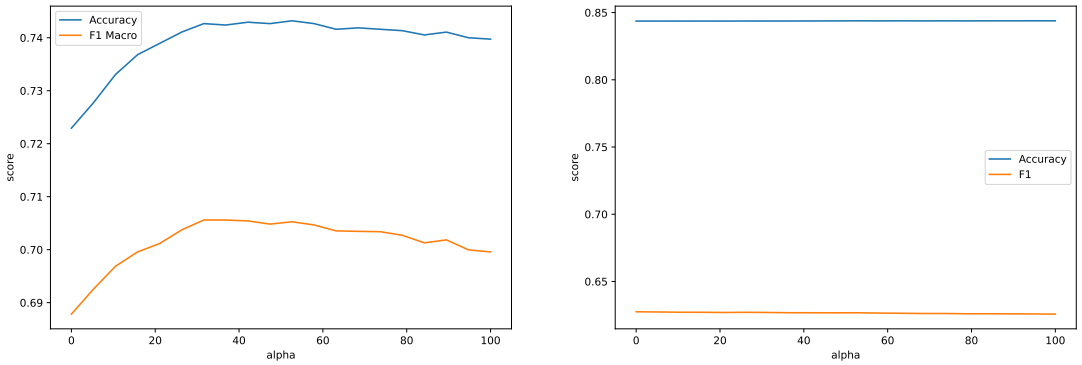


Figure 4: Sensitivity of the parameter `C` for the dataset "diabetes"

Ridge Classifier: Studying the sensitivity plots for this classifier, we can established that the influence of the hyperparameter `alpha` on the effectiveness scores varies by dataset. In figure 5 we show the two most extreme examples: For the dataset "Amazon Commerce Reviews" the results are definitely dependent on `alpha`, whereas for the dataset "Census Income" accuracy and F1 score hardly change.



(a) "Amazon Commerce Reviews"

(b) "Census Income"

Figure 5: Sensitivity of the parameter `alpha`

The parameter `fit_intercept` always influenced the result, but the magnitude varied depending on the dataset. Finally, in figure 6 we visualize the effect on performance measures for the dataset "diabetes" when changing the parameter `alpha` for both values of `fit_intercept`.

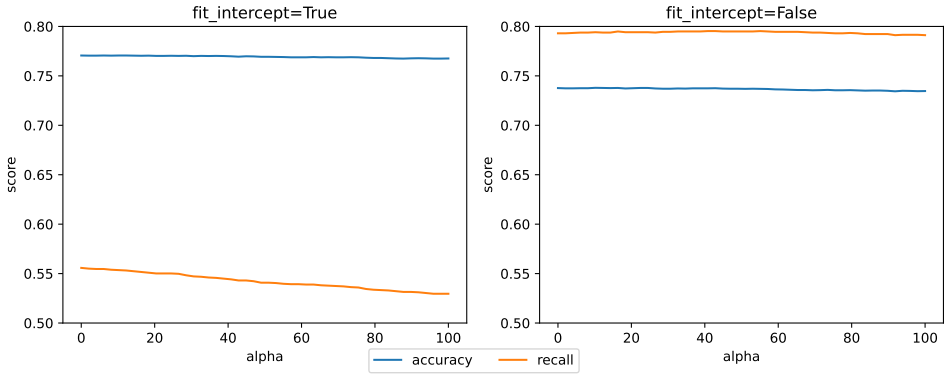


Figure 6: Sensitivity of the parameter `alpha` for the dataset "diabetes"

During the parameter tuning we couldn't decide on the best setting for this dataset. Because we prefer a higher recall over accuracy, it is clear from figure 6, that we have to set `fit_intercept` to 'False'. In this case we get the best result for both metrics when `alpha` = 40. So these are the parameters we would recommend for this dataset, giving us an accuracy of 0.737 and a recall of 0.795.

6 Results and Findings

The goal of this section is to compare the classifiers for each dataset and give a general overview on the performance of the tested classifiers. To make this comparison fair, we ensured to use exactly the same splits within each dataset in the cross-validation. This is achieved by always using the same `random_state` when creating the splits. Of course we applied the most successful preprocessing steps as discussed in section 4 and the hyperparameter settings established in section 5, which ensures that each classifier performs as good as possible. This way we can accurately state which classification algorithm performed best for each dataset.

In table 11 we summarized the top performance measures for each dataset and classifier. We also compared the best result for accuracy with the accuracy of the "Dummy classifier", which is shown in table 11. The "Dummy classifier" always predicts the majority class, so it can be used to establish a baseline. If the accuracy of a classifier is better than this baseline, then that classifier must have learned something. We can only use accuracy as a metric for the "Dummy classifier", because the F1 score is not defined as you would divide by zero in the calculation.

	congressional voting		Amazon Reviews		Census Income		diabetes	
Score	Accuracy	F1	Accuracy	F1	Accuracy	F1	Accuracy	Recall
Random Forest	0.973	0.966	0.708	0.667	0.805	0.677	0.766	0.623
LinearSVC	0.969	0.961	0.742	0.71	0.866	0.684	0.736	0.796
Ridge	0.968	0.959	0.743	0.706	0.844	0.626	0.737	0.795
Baseline	0.624	-	0.031	-	0.761	-	0.651	-

Table 11: Best results for each dataset and classifier

From this table we can conclude that on the dataset "congressional voting" the random forest classifier performed the best. If we again prefer F1 score for the dataset "Amazon Commerce Reviews", then the linear support vector classifier gives the best results. For the dataset "Census income" we can see that the linear support vector classifier is performing best with respect to both performance measures. Lastly, for the "diabetes" dataset there is no clear choice. But if we keep our preferences the same as in the last few sections, we can conclude that either the linear support vector classifier or the ridge classifier are the best choice for this dataset. The scores in table 11 are computed using repeated stratified 5-fold cross-validation. We chose 10 repetitions for the datasets "congressional voting" and "diabetes" and 5 repetitions for the other two datasets (due to runtime).

At this point we also want to compare cross-validation to the hold-out method. We did this by plotting box-plots for each effectiveness measure, dataset and classifier, displaying the calculated scores during cross-validation and their mean. Within each boxplot we also visualized the score of one hold-out split, where 70% of data were used for training and the rest for testing. The scores for the hold-out method

highly depend on the split chosen as seen in figures 7 and 8, where the box-plots are shown for the datasets "congressional voting" and "Amazon Commerce Reviews". In contrast, for the dataset "Census Income" the hold-out scores are almost the same as the mean of the cross-validation, which is to be expected because this dataset has a lot of instances. The box-plots for this dataset (figure 9) are also comparatively very narrow, meaning that the scores calculated during cross-validation vary very little i.e. the variance is really low.

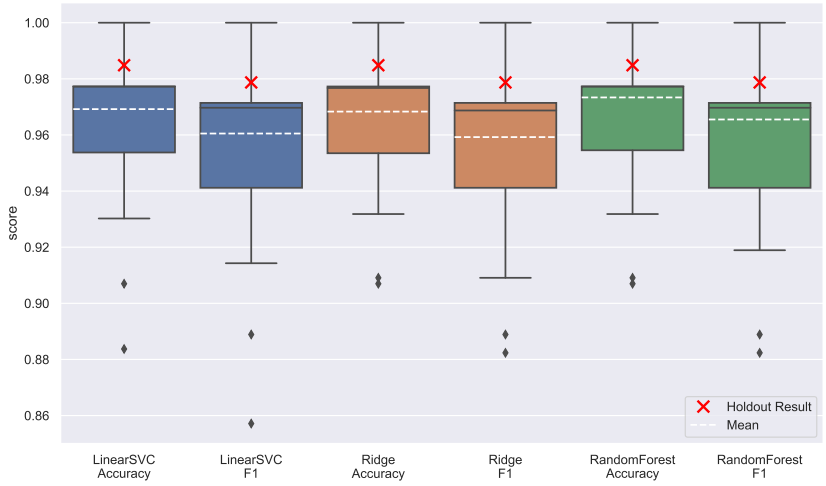


Figure 7: Model comparison for the dataset "congressional voting"

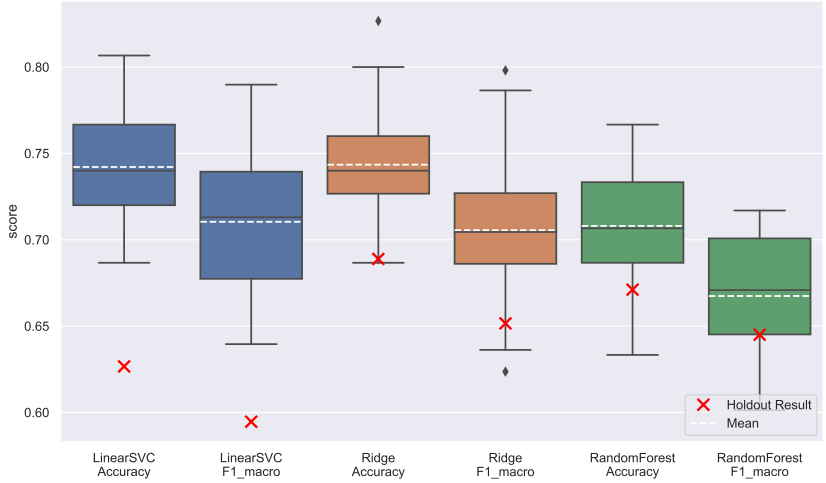


Figure 8: Model comparison for the dataset "Amazon Commerce Reviews"

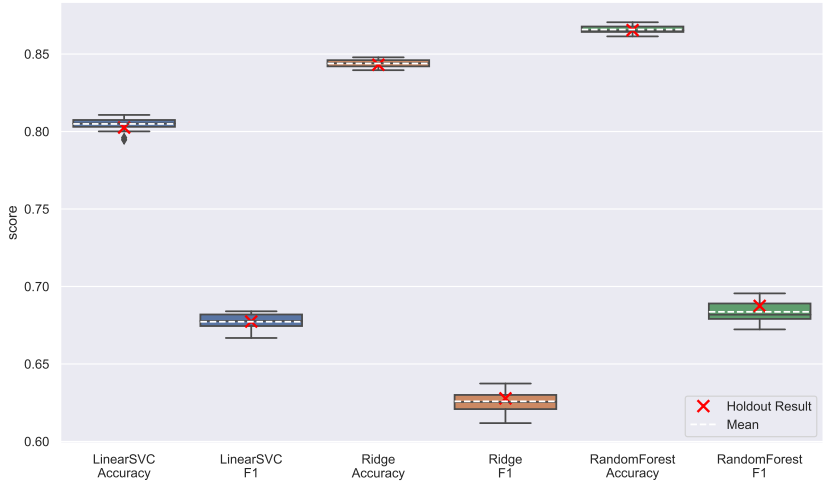


Figure 9: Model comparison for the dataset "Census Income"

We also want compare the efficiency of the classifiers. For this we used the runtime of the hold-out method calculation from above. We measured both the time for training (including preprocessing) and the time it took the test the model. Those values are displayed in table 12.

	congressional voting		Amazon Reviews		Census Income		diabetes	
Runtime	Training	Test	Training	Test	Training	Test	Training	Test
Random Forest	2.314	0.193	22.718	0.419	189.797	3.396	5.962	0.3388
LinearSVC	0.005	0.002	1.144	0.041	0.381	0.041	0.005	0.0016
Ridge	0.006	0.002	2.188	0.07	0.164	0.046	0.012	0.0014

Table 12: Runtime (in seconds) for each classifier and dataset combination for the hold-out method with a 70/30 split

It is evident that random forest was the slowest classifier we tested both for training and testing by a huge margin. On the contrary, the linear support vector classifier and ridge classifier performed similar on the first glance. The results from "Amazon Commerce Reviews" and "Census Income" might suggest, that the linear support vector classifier is faster for high dimensional datasets, whereas the ridge classifier is quicker with a high number of instances. We found out that the parameter `dual` of the linear support vector classifier has a great influence on its runtime. This parameter describes whether the algorithm should solve for the primal or dual optimization problem. Let n be number of samples and d the number of dimensions of a dataset. The dual formulation requires the computation of a $n \times n$ matrix, whereas in the primal formulation a $d \times d$ matrix is calculated. Therefore the primal formulation is faster for datasets, where $n > d$, which was especially noticeable for the "Census Income" dataset.

After all experiments our final conclusion is that no classifier we used outperformed the other across all datasets. The results in table 11 imply that the effectiveness of a classifier is highly dependent on the dataset. What we can establish is that the linear support vector classifier was never the worst in both metrics. Therefore it seems to be a very versatile classifier. In comparison the success for random forests varied quite a lot for different datasets and effectiveness measures. This could be due to the fact, that decision trees generally tend to overfit the model. As for efficiency, table 12 highlighted a big weakness of the random forest classifier, namely that it's comparatively very slow. Summing up section 4, choosing the right preprocessing steps, especially the scaler and dimension reduction, influences the results notably, though in varying magnitude for different classifiers and datasets. Here random forest seems the be the least influenced by the choice of scaler as seen in table 4 and 5. The sensitivity analysis in section 5 showed how much of an impact even a single hyperparameter can have on the overall performance of a model. However, comparing the results of section 4 with the ones in section 5 and table 11, it is clear that tuning the parameters hardly influenced the result for the "congressional voting" dataset. In contrast, the recall score for the "diabetes" dataset and both effectiveness scores for the "Amazon Commerce Reviews" improved a lot after the parameter tuning. For the dataset "Census Income" only the support vector classifier improved after optimizing the F1 score. As a summary of this whole project we would say that the best strategy for a classification problem majorly depends on the dataset in the beginning. Once a classifier is chosen everything else depends on the combination of the two including the importance of parameter tuning and preprocessing as well as the overall success.