# Machine Learning Exercise 1: Classification Report

**Group 20**: *Armin Nikbakht Tehrani, 11928288; David Deutz, 52104663; Marko Georgeiv, 12442103*

## Overview

This report serves as a resume of our activities during the completion of Exercise 1: Classification from the Machine Learning course at TU Wien. The exercise tasks us with working with a total of 4 datasets and 3 different categories of classifiers, to produce meaningful results regarding metrics that we will set ourselves. Our report will first describe the datasets used and why they were chosen. Then we will cover the specific classifiers and their categories with our predictions for the results. Furthermore, we will look at the task activities, focusing on data preprocessing and model performance, along with the metrics we chose to focus on and why. We will also compare our models in different circumstances and with different parameters. We will look at how changing the train-test split and the number of folds in cross-validation affects a base model, and then we will create models with different parameters, which we will test on one of these splits. We will finish with a final comparison between model categories and a conclusion.

## Dataset Description

We will start by describing the datasets we chose for the exercise. We will cover their dimensions. Then, we move on to the analysis of their features and target attributes. We will also briefly go over the overall data structure and the source of the dataset.

### Congressional Voting

The **Congressional Voting** dataset aims to predict the political party a congress member is associated with based on past voting decisions on selected topics. The dataset has a total of 218 instances and 18 attributes. Each feature has a categorical data type and can assume the value ‘y’ or ‘n’. The target feature also has a categorical data type and can assume the value “republican” or “democrat”, i.e., it's a binary classification task. Of those 218 instances, 103 instances are missing a value for at least one feature, which is characterized by the value being “unknown” instead of “y” or “n”. This dataset was assigned to us based on the parity of our group number.

### Amazon Reviews

The **Amazon Reviews** dataset aims to show the number of times a certain word/tag has been used in a review left by someone on a product on Amazon. Based on the number of words, the idea is to see who left the review, so the target class contains the names of the people leaving the reviews. The dataset has 750 instances and 10,000 attributes. The dataset is extremely high-dimensional and very sparse, containing mostly zeroes. As we mentioned, these are very likely just counters for specific words that have been abstracted away. For example, the column in V1 could be the word ‘the’, and the values for that column are how many times each reviewing user used that word in a review. This dataset was assigned to us based on the parity of our group number. In total, the target attribute, i.e., the reviewing user name, has a total of 50 possible values.

### Airplane Passenger Satisfaction

The **Airplane Passenger Satisfaction** dataset aims to predict the satisfaction of an airplane passenger based on the passenger's personal details and further information on airplane service and delays.

The dataset has 129,879 instances and 24 features. The target variable is again category-based with two possible values: *satisfied* and *neutral, or dissatisfied*. Thus, the task for this dataset is a binary classification problem. The dataset has already been split into training and a test set by the contributor, whereby the training data contains 103,904 instances while the test data consists of the remaining 25,975 instances. The training data contains a small number of missing values, for which we provide an explanation on how we dealt with them later in the report. We chose this dataset to add variety to our choices, as we needed one dataset with many instances, relative to other datasets in our collection.

### Students' Dropout and Academic Success

The **Student's Dropout and Academic Success** dataset aims to predict the dropout and academic success of students from a higher education institute at the end of a course based on information known at the time of the student's enrollment, such as *demographics*, *socio-economic factors,* and *academic paths*. The dataset has 4,424 instances and 36 features, all of which are either interval or ratio scales. The target variable is category-based with three distinct values: *Dropout*, *Enrolled,* and *Graduate*. Thus, the task for this dataset is a three-class classification problem.

The data contributor on Kaggle has already encoded all features. For instance, the *Marital Status* variable has been encoded as follows: 1 - single, 2 - married, 3 - widower, 4 - divorced, 5 - facto union, 6 - legally separated. Figure 1 shows the distribution of the target column. We can see that there might be a slight imbalance between the classes during model training, which could potentially cause issues and might have to be dealt with. Furthermore, there might be a certain skewness in the numeric variables of the dataset, as can be seen in Figure 2 below, which depicts the distribution of values within the *Age at enrollment* feature. This dataset was the one we chose from Exercise 0.

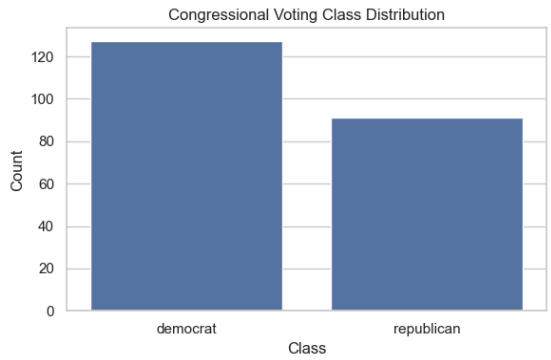


Figure 1

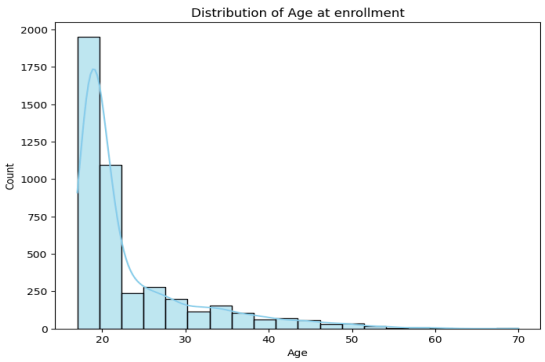


Figure 2

## Classifiers Overview

Now, we will look at the three different classifiers we chose for this exercise. We will cover which classifier category they belong to, along with their general characteristics in regard to our datasets (without repeating what was already covered in the lecture). We will also make general predictions on how these models will do, considering the dataset descriptions provided earlier.

### Overview of the K-Nearest Neighbor Classifier

The first classifier that we chose was the **k-Nearest Neighbor Classifier (KNN)**, which is a *distance-based* classifier. It determines the membership of an observation according to the *k* nearest observations in a majority voting manner. For instance, if three out of the five nearest neighbors of the new observation belong to class A, then the new observation will be classified as class A. We chose this classifier mainly because it is one of the simplest methods and can easily be compared to more complex machine learning algorithms in terms of performance. Furthermore, we were planning on experimenting with different distance metrics that can be used to determine the neighborhood of an observation, as well as the number of neighbors to look at. Since it determines the membership based on a distance metric, KNN is very sensitive to the scale and distribution of the data, which we have to consider when training a model using it. However, it provides another possibility to experiment, as we can measure the time it takes KNN to fit the data and predict the test sets of the data.

### Overview of the Random Forest Classifier

The second classifier that we chose was the **Random Forest Classifier (RFC)**, which is an *ensemble-type* classifier, namely, it is an ensemble of decision tree classifiers. This classifier was chosen as it's a versatile classifier that is not impacted by the scale of feature values, i.e., value normalization has no impact, and because this classifier is well suited to handle high-dimensional data as it does not depend on all features at once. In the scope of this exercise, we are planning to experiment with different Random-Forest hyperparameters, e.g., changing the number of trees or changing the criterion used to compute the best possible split for each node, and observing the impact those changes have.

### Overview of the Multi-Layer Preceptor

The last classifier we chose was a **Multi-Layer Perceptron (MLP)** as a representative of the *Neural Network type*. An MLP is a good choice for these exercises, as it is suitable for modeling nonlinear relationships in structured tabular data. MLPs are notorious for being inaccurate when dealing with high-dimensional data. Since the Amazon Review dataset is 750 by 10,000, this leads to interesting results from the MLP. Following this logic, we predict that MLPs will perform very well for datasets with many entries, without too much tweaking, provided we fine-tune the number of epochs and learning rate. However, datasets with fewer entries will likely fall behind the other models. We will experiment with adding and removing layers, adjusting learning rates, epochs, and splits, and we will see how these factors will affect our target metrics.

## Task Completion

This section will cover the tools and general methodology our team followed during the completion of this exercise. We will cover data importing and the preprocessing required by each of the three classifiers on all four datasets.

### Data Importing

The data was imported using *pandas read_csv()* function. In the .zip folder of this submission, we provided the folder structure that we prepared for the notebooks to ensure that they can be run by anyone who downloads them.

## Data Preprocessing

Data preprocessing was a major part of this exercise. However, it was vastly different across the classifiers we chose. In this section, we will cover what data preprocessing was used for each classifier.

### *Tools Used For Every Classifier*

Even though the preprocessing was different, some tools were used across all models. To avoid repetition later, we will look at them now. For scaling, both KNN and MLP relied on the **StandardScaler()**, which standardizes the input data by removing the mean and scaling it to unit variance. We test MLP only using this scaler, but for KNN, we tried different scaling approaches. Moreover, across all models, for encoding, we only relied on **LabelEncoder()** from *sklearn* to transform categorical variables and target labels into integer values

### *k-Nearest Neighbours (KNN) Preprocessing*

As already mentioned, KNN is very sensitive to the scaling and distribution of the data, because of its distance-based nature. Therefore, we had to ensure that the data was appropriately preprocessed before starting with the model training. The following paragraphs will provide a more detailed description of the steps that were performed for each of the four datasets.

The scaling of the data was carefully evaluated using different methods provided by scikit-learn. Hereby, 10-fold cross-validation was used to establish a pipeline with an appropriate scaling method to be applied to the data before the model is trained. We mainly focused on three scaling methods from scikit-learn. Firstly, the **MinMaxScaler()**, which scales the input data between a given range, we also tried **RobustScaler()**, a robust method against outliers, which is especially important for KNN, and we also used the **StandardScaler().** As done with the scaling, the feature selection process was also evaluated through 10-fold cross-validation using different strategies from scikit-learn. Hereby, we focused again on three different methods. We tried **VarianceThreshold()**, which removes features with low variance. We also used **SelectKBest()**, an approach that selects the k best features according to some scoring function (ANOVA F by default). Lastly, we used **Principal Component Analysis (PCA).** Further preprocessing steps, like the encoding of categorical features and the imputation of missing values, were also conducted using built-in scikit-learn objects.

### *Random Forest Classifier (RFC) Preprocessing*

Firstly, as the sklearn Random Forest classifier implementation is not able to handle categorical feature values, an important preprocessing step is to transform categorical feature values to numerical values. This was only necessary for the “Airplane Passenger Satisfaction” and the “Congressional Voting” datasets, as the remaining datasets already contained only numerical values for the features.

For the “Congressional Voting” dataset, we encoded the categorical value ‘n’ as ‘0’ and the value ‘y’ as 1. Since Random Forest classifiers are indifferent to the actual value (i.e., if the value is 1 or 2 or 3, etc.) it is not necessary to experiment with different values. For the “Airplane Passenger Satisfaction”, we needed to encode the values for the “Gender”, “Customer Type”, “Type of Travel”, and “Class” features. As already mentioned, we used **LabelEncoder()** to encode the data.

The **second important preprocessing step** that was necessary when training a Random Forest classifier was dealing with missing values. This was necessary for the “Congressional Voting” and the “Airplane Passenger Satisfaction” datasets, as those two were the datasets that contained missing

values. To handle the missing values for the "Congressional Voting" dataset, we tried two approaches. The first approach we took was replacing the value "unknown" (representing a missing value in this dataset) with the numeric value "3", effectively introducing a new category. The second approach we took was using sklearn's **SimpleImputer()** to infer a possible value if the value is missing.

When handling missing values for the "Airplane Passenger Satisfaction" dataset, we also went with two approaches. Since there are over 100k instances, and only about 300 of them have missing values, the first approach was simply removing these instances. The second approach we took was using *sklearn's* **SimpleImputer()** to infer a possible value based on the values that other instances in the dataset have for this feature. As already noted in the classifier description section, an RFC is not impacted by scaling and distribution of data, so no further preprocessing steps were necessary.

### *Multi-Layer Perceptron (MLP) Preprocessing*

In the context of the MLP, the raw data needed to be preprocessed for every dataset, in some way. Preprocessing for the MLP is fairly straightforward. MLPs require numerical, fixed-size input. So, the preprocessing is composed of encoding categorical values, handling missing ones, normalization and standardization where needed, and converting labels into integer types. Across all datasets, we only relied on **StandardScaler()** for scaling. The only exception was the "Congressional Voting" dataset, which, due to the encoding approach, didn't need to scale. In terms of encoding, just like with all our classifiers, we used **LabelEncoder()**. It was applied to both input features (like "Gender" or "Customer Type") and output labels (like "satisfied", "Dropout", etc.). In the "Congressional Voting" dataset, vote choices "y", "n", and "unknown" were directly mapped to 1, 0, and 0.5, respectively, preserving their semantic meaning numerically. Implicitly handling the missing values as a new in-between category. For missing values in the "Airplane Passenger Satisfaction" dataset, we didn't want to remove entries, as the MLP performance requires as many of them as possible, so we substituted them with zero. For scaling, everywhere where we had to use scaling for the MLP, we went with the StandardScaler, as described earlier. Lastly, feature arrays were converted to float32 tensors and label arrays to int64 (long) tensors for compatibility with PyTorch.

## Target Metrics

For the performance measurements of this exercise, we chose to measure ***accuracy***, ***precision***, ***recall***, ***and F1-score***. In this section, we will rationalize our choices and show why these target metrics were the right choice. The **accuracy** measures how often the machine learning model correctly predicts the class membership of observations. We chose this measurement to have a general indicator of the model's performance. Since we did not experience any large imbalances in the classes of the data, we chose to use the traditional accuracy instead of the **Balanced Accuracy**.

The **precision** of a machine learning model measures how often the positive class is correctly predicted. We chose this metric to get a slightly better insight into the model's performance, also in regard to the slight imbalance of classes in some cases, than we get with the accuracy measure. The **recall** measures how often the machine learning model identifies positive instances of the data. We also chose this metric since it is more suitable for imbalanced classes and to get a slightly better insight into the model's performance.

To have a reliable measurement, especially for multi-class problems, we chose to use the **F1-score** as our last performance metric, which is the harmonic mean of precision and recall. Lastly, it's worth

mentioning that we also often used the **classification report** object of scikit-learn to evaluate the performance of our models.

# Model Performances

For the model performances, after a lengthy discussion, we finally decided on our approach. We will review a total of three train/test splits (50/50, 70/30, 90/10) and two cross-validation configurations (one with 5 and another with 10 folds). Across all datasets and evaluation setups (fixed train/test splits and cross-validation), we clearly observe differences in performance metrics (accuracy, precision, recall, and F1-score) both across datasets and within the same dataset under different data split configurations. These variations are natural and are influenced by several interrelated factors: dataset size, class distribution, model complexity, and evaluation strategy.

## The k-Nearest Neighbours (KNN)

The following section will provide some detailed reports of the classifiers' performance along the process of finding the best model. As can be seen in the notebook, we decided to start with a baseline classifier without any preprocessing steps except data imputation if necessary. Then, we proceeded with adding the preprocessing steps needed to hopefully boost the performance of our classifiers.

| KNN | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Splits | Amazon Reviews | | | | Students' Dropout and Academic Success | | | | Airplane Passenger Satisfaction | | | |
| | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score |
| **50/50** | 0.2960 | 0.3082 | 0.3560 | 0.2757 | 0.7486 | 0.6530 | 0.6956 | 0.6620 | 0.9354 | 0.9307 | 0.9380 | 0.9337 |
| **70/30** | 0.3378 | 0.3346 | 0.3831 | 0.3054 | 0.7681 | 0.6640 | 0.7286 | 0.6743 | 0.9795 | 0.9781 | 0.9801 | 0.9790 |
| **90/10** | 0.3378 | 0.3346 | 0.3831 | 0.3054 | 0.7681 | 0.6640 | 0.7286 | 0.6743 | 0.9385 | 0.9343 | 0.9404 | 0.9369 |

Table 1

The baseline KNN model on the **Student's Dropout and Academic Success** dataset had an accuracy of about 65,91%. After adding an appropriate scaling and feature selection method, the final classifier achieved the performances shown in Table 1 (the different holdout splits). Thus, we managed to boost the accuracy of the test data by over 70%, which is a reasonable performance. We can see that the performances of the KNN models, which were trained on the 70/30 and 90/10 splits, are slightly better. The reason behind this is likely the higher amount of data to train on. Interestingly, the performance measurements are exactly the same across all metrics for the 70/30 and 90/10 splits.

On the **Amazon Review** dataset, the baseline KNN model had an accuracy of ~32,24%, which is not great at all. In our opinion, KNN is not suitable for the nature of the dataset, which leads to the performance of the model. Nevertheless, we applied a scaling and feature selection method and achieved the following performances on the different holdout splits. Although we managed to boost the performance a little bit, it is still not reasonable at all, which is why we think that the underlying problem cannot be solved with a KNN and likely requires more advanced machine learning algorithms or feature engineering methods. Nevertheless, it is again noteworthy that the 70/30 and the 90/10 splits have the exact same performance on the test data in all metrics.

The **Airplane Passenger Satisfaction** dataset proposed a unique challenge in terms of runtime since the split that contained 70% and 90% of the training data took 80 minutes for the model to fit. The baseline model had an accuracy of ~80.9%, which is a reasonable performance. After scaling and feature selection, we managed to achieve the following performances on the different holdout splits. We can see that the model generalizes very well and has good performance on all metrics and across

all holdout splits. However, it is noteworthy that KNN might not be the best choice for this dataset due to its high number of rows. Since KNN is lazy and only evaluates the neighbors during prediction, the prediction for the test set also took a very long time, which might be a factor to consider when choosing a model.

### *Congressional Voting*

In the **Congressional Voting** data, we chose to compare two alternative methods to handle the missing values. In the first approach, we impute them through the most frequent value within the feature; the other option is to treat them as a third category. Both results can be seen in Table 2. The baseline model that was trained on the data with the imputed missing values outperformed the baseline model that was trained on the data with the missing values encoded as a third category with an accuracy of ~94,45% against ~92,64%. After scaling and feature selection, we achieved the following performances on the test sets of the different holdout splits.

| | Imputed | | | | Third Category | | | |
|---|---|---|---|---|---|---|---|---|
| Splits | Congressional Voting | | | | Congressional Voting | | | |
| | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score |
| **50/50** | 0.9537 | 0.9528 | 0.9542 | 0.9534 | 0.9352 | 0.9341 | 0.9355 | 0.9347 |
| **70/30** | 0.9538 | 0.9552 | 0.9508 | 0.9527 | 0.9385 | 0.9475 | 0.9355 | 0.9377 |
| **90/10** | 0.9091 | 0.8952 | 0.8952 | 0.8952 | 0.9545 | 0.9667 | 0.9375 | 0.9494 |

Table 2

We can see that the first missing value strategy, where we imputed the data outperformed the second strategy on the 50/50 and the 70/30 split. However, on the 90/10 split, the second strategy proved to be more efficient.

### *Influence of Model Architecture on Performance*

To determine the impact of different distance metrics on the performance of KNN, we decided to conduct an experiment on the **Congressional Voting** dataset. Therefore, we trained four classifiers with different distance metrics that are provided by scikit-learn: **cosine**, **euclidean**, **manhattan,** and **cityblock**. We ensured a fair comparison by setting the remaining hyperparameters to the same values for each of the classifiers. The performance results of those classifiers can be found in the table below.

| Distance Metric | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| **Euclidean** | 0.6000 | 0.5130 | 0.5197 | 0.5143 |
| **Manhattan** | 0.6689 | 0.5755 | 0.5928 | 0.5794 |
| **Cosine** | 0.6305 | 0.5320 | 0.5523 | 0.5352 |
| **Cityblock** | 0.6689 | 0.5755 | 0.5928 | 0.5794 |

Table 3

The results show that Manhattan and Cityblock distances yield the highest accuracy (≈66.89%), precision (≈57.55%), recall (≈59.28%), and F1-score (≈57.94%), indicating that they performed best overall. Cosine distance followed with moderate performance across all metrics, while Euclidean

distance resulted in the lowest scores, with an accuracy of 60% and an F1-score of 51.43%. These findings suggest that the choice of distance metric significantly influences KNN effectiveness, with Manhattan-based metrics outperforming the others on this particular dataset and in this particular setting.

## The Random Forest Classifier (RFC)

Next, we will look at the performance of the Random Forest Classifier. Here the approach for cross-validation and holdout remains the same. First, we will show the characteristics of the models we used for the 'Holdout & Cross-Validation Evaluation'

### *Holdout & Cross-Validation Evaluation*

To find the best-performing model for each dataset and evaluation method, we performed a GridSearch over a selection of hyperparameter combinations. The obtained hyperparameter configurations are shown in Table 4, while Table 5 showcases the results that we got when computing the 50/50, 70/30, and 90/10 holdout evaluations and the 5- and 10-fold cross-evaluations with those models.

| Random Forest | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Congressional Voting | | | | Amazon Reviews | | | | Airplane Passenger Satisfaction | | | | Students' Dropout and Academic Success | | | |
| Splits | n_estimators | criterion | max_depth | max_features | n_estimators | criterion | max_depth | max_features | n_estimators | criterion | max_depth | max_features | n_estimators | criterion | max_depth | max_features |
| **50/50** | 50 | gini | 5 | sqrt | 400 | gini | 10 | sqrt | 200 | gini | 10 | sqrt | 150 | gini | 10 | sqrt |
| **70/30** | 100 | entropy | 5 | sqrt | 400 | gini | 10 | sqrt | 200 | gini | 10 | sqrt | 150 | gini | 10 | sqrt |
| **90/10** | 50 | entropy | 10 | sqrt | 400 | gini | 10 | sqrt | 50 | gini | 10 | sqrt | 150 | gini | 10 | log2 |
| **CV-5** | 50 | gini | 10 | sqrt | 400 | gini | 10 | sqrt | 200 | gini | 10 | sqrt | 50 | entropy | 10 | log2 |
| **CV-10** | 50 | entropy | 10 | sqrt | 400 | gini | 10 | sqrt | 50 | gini | 10 | sqrt | 100 | entropy | 10 | log2 |

Table 4

| Random Forest | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Congressional Voting | | | | Amazon Reviews | | | | Airplane Passenger Satisfaction | | | | Students' Dropout and Academic Succes | | | |
| Splits | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score |
| **50/50** | 0.9725 | 0.9700 | 0.9733 | 0.9715 | 0.5147 | 0.4996 | 0.5999 | 0.4829 | 0.9467 | 0.9447 | 0.9467 | 0.9456 | 0.7758 | 0.6752 | 0.7381 | 0.6865 |
| **70/30** | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 0.6622 | 0.6367 | 0.6944 | 0.6177 | 0.9460 | 0.9441 | 0.9460 | 0.9450 | 0.7703 | 0.6707 | 0.7206 | 0.6806 |
| **90/10** | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 0.6800 | 0.6800 | 0.6327 | 0.6295 | 0.9490 | 0.9470 | 0.9490 | 0.9479 | 0.7833 | 0.6802 | 0.7285 | 0.6872 |
| **CV-5** | 0.9679 | 0.9676 | 0.9678 | 0.9669 | 0.6333 | 0.6601 | 0.6160 | 0.5942 | 0.9463 | 0.9465 | 0.9442 | 0.9453 | 0.7728 | 0.7359 | 0.6675 | 0.6773 |
| **CV-10** | 0.9680 | 0.9685 | 0.9673 | 0.9668 | 0.6360 | 0.5746 | 0.6240 | 0.5691 | 0.9465 | 0.9467 | 0.9443 | 0.9454 | 0.7726 | 0.7379 | 0.6673 | 0.6780 |

Table 5

The "Congressional Voting" dataset is relatively small and well structured, resulting in the Random Forest Model delivering very good results across all measured performance metrics. From Table 4, it can be seen that the holdout performance scores do not fluctuate significantly when using different data split ratios, and the same goes for computing the cross-validation scores with different fold sizes. Furthermore, the performance scores computed using holdout and cross-validation do not differ significantly. However, it is worth noting that the accuracy scores are very high, and this was even the model that tied us for second, third, and fourth place in the competition.

For the "Amazon Reviews" dataset, the RF model delivers moderate results across all measured performance metrics. When looking at the computed holdout scores, we can see a considerable performance improvement when increasing the size of the training set. A possible explanation for this behaviour is that since the dataset contains a very large number of features, a training split can cause either underfitting or overfitting, consequently reducing the model's performance on unseen data. When looking at the cross-validation scores, we can observe that the performance scores are similar to the holdout scores received for the 90/10 split and that they remain relatively stable for the different fold sizes, showcasing that cross-validation can help to prevent under- or overfitting, though it comes with additional computational complexity.

For the "Airplane Passenger Satisfaction" dataset, we manage good results across all performance metrics for both the holdout and the cross-validation evaluation methods. We can see that the performance scores barely fluctuate when changing the split or fold size, and that the cross-validation scores are almost equal to the holdout scores. A possible reason for this observed stability when using different split and fold sizes is the fact that the dataset contains over 100k instances and only ~20 features, which reduces the significance of the split ratio.

For the "Students' Dropout and Academic Success" dataset, we manage moderate performance scores across all performance metrics, for both holdout and cross-validation. We can observe that the holdout performance scores do not differ significantly when using different split sizes and that the cross-validation scores are very similar for the different fold sizes. Comparing the holdout and cross-validation scores reveals that with holdout, we receive precision scores that are slightly lower than the recall score, and for cross-validation, it is exactly the other way around; we get higher precision scores but lower recall scores.

The runtimes varied greatly depending on the dataset and the performed task. The holdout evaluation was completed in less than 2 minutes for each dataset and each of the three data splits. The cross-validation for fold sizes 5 and 10 was also completed for each dataset in less than 2 minutes. The greatest runtimes were observed when performing the GridSearch for the hyperparameter tuning. When allowing unlimited tree depth for the "Airplane Passenger Satisfaction" and the "Amazon Reviews" the runtime far exceeds 2 hours, which is why we decided to limit the depth of the trees. With those limits, the runtime is reduced to less than 10 minutes.

## *Handling Missing Values*

In the "Congressional Voting" and the "Airplane Passenger Satisfaction" datasets, we have missing values, and in this section, we will analyze how different strategies to handle missing values affect the performance of the models.

### Congressional Voting

Since the features that are missing values are of categorical type and can only assume the values "y" and "n", the first strategy we used to deal with missing values was introducing a third category that represented a missing value. The second strategy we used was imputing the missing values using sklearn's **SimpleImputer** with the "most_frequent" strategy. For each of those two strategies we performed a GridSearch on the training data of the three splits (50/50, 70/30, and 90/10) and achieved the following results:

| | Third Category | | | | Imputing missing values | | | |
|---|---|---|---|---|---|---|---|---|
| **Splits** | **Accuracy** | **Precision** | **Recall** | **F1-Score** | **Accuracy** | **Precision** | **Recall** | **F1-Score** |
| **50/50** | 0.9633 | 0.9589 | 0.9656 | 0.9619 | 0.9725 | 0.9700 | 0.9733 | 0.9715 |
| **70/30** | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| **90/10** | 0.9545 | 0.9444 | 0.9643 | 0.9521 | 1,0000 | 1,0000 | 1,0000 | 1,0000 |

Table 6

As can be seen in Table 6, imputing missing values yields slightly better results than introducing a third category. When introducing a third category, the performance decreases when going from a 70/30 to a 90/10 split, which could be caused by the model learning patterns that do not actually exist in the data, but were introduced by the missing-value handling strategy.

### Airplane Passenger Satisfaction

Since this dataset contains over 100k instances and only about 300 of them were missing values, our first strategy was simply deleting those instances from the set. The second strategy we used was again imputing the missing values using sklearn's **SimpleImputer** with the "most_frequent" strategy. For both strategies, we performed a GridSearch on the training data of the three data splits and achieved the following results:

| | Delete instances w/ missing values | | | | Imputing missing values | | | |
|---|---|---|---|---|---|---|---|---|
| **Splits** | **Accuracy** | **Precision** | **Recall** | **F1-Score** | **Accuracy** | **Precision** | **Recall** | **F1-Score** |
| **50/50** | 0.9467 | 0.9447 | 0.9467 | 0.9456 | 0.9444 | 0.9423 | 0.9443 | 0.9433 |
| **70/30** | 0.9460 | 0.9441 | 0.9460 | 0.9450 | 0.9446 | 0.9424 | 0.9448 | 0.9435 |
| **90/10** | 0.9490 | 0.9470 | 0.9490 | 0.9479 | 0.9456 | 0.9432 | 0.9461 | 0.9445 |

Table 7

As can be seen in Table 7, the strategy employed to handle missing values barely has any impact on the performance of the trained model. This can be explained by the fact that only about 300 of over 100k instances are actually missing values, and therefore, those strategies do not impact the learning of the model.

### *Influence of Model Architecture on Performance*

To assess how changes in a Random Forest's hyperparameters impact the model performance, we chose for each dataset the model that across all three data splits produced the best results, and used them as baseline to examine how modifying selected hyperparameter values affects performance.

| | | Congressional Voting | | | |
|---|---|---|---|---|---|
| Classifiers | Classifier Description | Accuracy | Precision | Recall | F1-Score |
| Baseline | n_estimators= 100<br>max_depth = 5<br>max_features = sqrt<br>criterion=entropy | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| RFC2 | n_estimators= 100<br>max_depth = 5<br>max_features = sqrt<br>criterion=gini | 0.9848 | 0.9868 | 0.9828 | 0.9846 |
| RFC3 | n_estimators=5<br>max_depth=5<br>max_features=sqrt<br>criterion=entropy | 0.9848 | 0.9821 | 0.9872 | 0.9844 |
| RFC4 | n_estimators=10<br>max_depth=5<br>max_features=sqrt<br>criterion=entropy | 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| RFC5 | n_estimators=400<br>max_depth=5<br>max_features=sqrt<br>criterion=entropy | 0.9848 | 0.9868 | 0.9828 | 0.9846 |
| RFC6 | n_estimators=600<br>max_depth=5<br>max_features=sqrt<br>criterion=entropy | 1,0000 | 1,0000 | 1,0000 | 1,0000 |

Table 8.A

| | | Amazon Reviews | | | |
|---|---|---|---|---|---|
| Classifiers | Classifier Description | Accuracy | Precision | Recall | F1-Score |
| Baseline | n_estimators=400<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.6800 | 0.6800 | 0.6327 | 0.6295 |
| RFC2 | n_estimators=400<br>max_depth=10<br>max_features=sqrt<br>criterion=entropy | 0.6133 | 0.6000 | 0.5483 | 0.5519 |
| RFC3 | n_estimators=15<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.3200 | 0.3200 | 0.2317 | 0.2580 |
| RFC4 | n_estimators=800<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.7467 | 0.7300 | 0.6747 | 0.6808 |
| RFC5 | n_estimators=1200<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.7467 | 0.7400 | 0.7047 | 0.6968 |

Table 8.B

| | | Airplane Passenger Satisfaction | | | |
|---|---|---|---|---|---|
| Classifiers | Classifier Description | Accuracy | Precision | Recall | F1-Score |
| Baseline | n_estimators=50<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.9490 | 0.9470 | 0.9490 | 0.9479 |
| RFC2 | n_estimators=50<br>max_depth=10<br>max_features=sqrt<br>criterion=entropy | 0.9480 | 0.9461 | 0.9481 | 0.9470 |
| RFC3 | n_estimators=5<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.9417 | 0.9396 | 0.9416 | 0.9405 |
| RFC4 | n_estimators=300<br>max_depth=10<br>max_features=sqrt<br>criterion=gini | 0.9497 | 0.9479 | 0.9497 | 0.9487 |

Table 8.C

| | | Students' Dropout and Academic Success | | | |
|---|---|---|---|---|---|
| Classifiers | Classifier Description | Accuracy | Precision | Recall | F1-Score |
| Baseline | n_estimators=150<br>max_depth=10<br>max_features=log2<br>criterion=gini | 0.7833 | 0.6802 | 0.7285 | 0.6872 |
| RFC2 | n_estimators=150<br>max_depth=10<br>max_features=log2<br>criterion=entropy | 0.7743 | 0.6688 | 0.7059 | 0.6719 |
| RFC3 | n_estimators=15<br>max_depth=10<br>max_features=log2<br>criterion=gini | 0.7720 | 0.6700 | 0.7073 | 0.6739 |
| RFC4 | n_estimators=400<br>max_depth=10<br>max_features=log2<br>criterion=gini | 0.7788 | 0.6745 | 0.7128 | 0.6795 |

Table 9.D

The "Congressional Voting" dataset, for which we can see the results in Table 8.A, is well structured and contains a relatively small number of features. Based on those facts we can conclude that the number of trees is not crucial to the model's performance. For 100 trees we achieve a score of '1' on all measured metrics, and when increasing or decreasing the number of trees, the performance scores barely fluctuate. The slight fluctuation that we see is most likely introduced by the randomness of e.g. feature selection and/or bootstrapping.

The "Amazon Reviews" dataset, for which we can see the results in Table 8.B, contains relatively few observations but a large number of features. Since the single decision trees are trained on only a subset of the data and a subset of the features, this fact alone already indicates that the number of trees is highly relevant to the model's performance, which we confirmed with our experiments. When decreasing the number of trees from 400 to only 15, the performance scores across all measured metrics are cut by more than half. When increasing the number of trees from 400 to 800, we observe a moderate increase across all performance scores. It is, however, noteworthy that increasing the number of trees even further results in diminishing returns, as, e.g., 1200 trees do not yield any noteworthy performance increase when compared to using 800 trees.

Both the "Airplane Passenger Satisfaction" and the "Student's Dropout and Academic Success" datasets, for which we can see the experiment results in Table 8.C and 8.D respectively, have a relatively low number of features considering the number of observations, which indicates that the number of trees is not crucial to the model's performance. This statement is supported by our experiments, which show that varying the number of trees causes almost no difference across all four measured performance metrics. Furthermore, since both datasets are imbalanced, the model exhibits similar performance scores for both the "gini" and the "entropy" split criterion.

## The Multi-Layer Perceptron (MLP)

Lastly, we will look at the Multi-Layer Perceptron model. Before getting into the results, it is worth noting that the execution of the MLP code was done locally, instead of on Google Colab. That is why, in contrast to the other models, it competed much quicker. That being said, let's get into the analysis.

### *Holdout & Cross-validation*

First, here are the baseline models for each dataset for this section of the report. We relied on the *torch.nn* module to create them. The parameters we have chosen are shown here in Table 9. Respectively, the results of the 50/50, 70/30, and 90/10 holdouts and the 5 and 10-fold cross validations when using the base models from Table 9 are shown in Table 10.

| Congressional Voting | Amazon Reviews | Airplane Passenger Satisfaction | Students' Dropout and Academic Success |
|---|---|---|---|
| nn.Linear(input_dim, 16),<br>nn.ReLU(),<br>nn.Linear(16, 8),<br>nn.ReLU(),<br>nn.Linear(8, 2), | nn.Linear(input_dim, 512),<br>nn.ReLU(),<br>nn.Dropout(0.3),<br>nn.Linear(512, 256),<br>nn.ReLU(),<br>nn.Dropout(0.3),<br>nn.Linear(256, output_dim) | nn.Linear(input_size, 128),<br>nn.ReLU(),<br>nn.Dropout(0.3),<br>nn.Linear(128, 64),<br>nn.ReLU(),<br>nn.Linear(64, 32),<br>nn.ReLU(),<br>nn.Dropout(0.3),<br>nn.Linear(32, output_size), | nn.Linear(input_size, 64),<br>nn.ReLU(),<br>nn.Dropout(0.3),<br>nn.Linear(64, 32),<br>nn.ReLU(),<br>nn.Linear(32, output_size) |

Table 9

| Multi-Layer Preceptron (MLP) | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Splits | Congressional Voting | | | | Amazon Reviews | | | | Airplane Passenger Satisfaction | | | | Students' Dropout and Academic Success | | | |
| | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score | Accuracy | Precision | Recall | F1-Score |
| 50/50 | 0.9358 | 0.8800 | 0.9778 | 0.9263 | 0.5120 | 0.6039 | 0.4932 | 0.4903 | 0.9590 | 0.9710 | 0.9336 | 0.9519 | 0.7622 | 0.7028 | 0.6834 | 0.6900 |
| 70/30 | 0.9848 | 1.0000 | 0.9643 | 0.9818 | 0.5644 | 0.6756 | 0.5533 | 0.5569 | 0.9606 | 0.9693 | 0.9391 | 0.9539 | 0.7508 | 0.6832 | 0.6707 | 0.6754 |
| 90/10 | 0.9545 | 1.0000 | 0.8889 | 0.9412 | 0.6400 | 0.5617 | 0.6200 | 0.5607 | 0.9601 | 0.9698 | 0.9374 | 0.9533 | 0.7675 | 0.7072 | 0.6860 | 0.6934 |
| CV-5 | 0.9680 | 0.9666 | 0.9676 | 0.9670 | 0.6280 | 0.6595 | 0.6200 | 0.5967 | 0.9648 | 0.9750 | 0.9433 | 0.9589 | 0.7547 | 0.6892 | 0.6741 | 0.6789 |
| CV-10 | 0.9634 | 0.9651 | 0.9621 | 0.9623 | 0.6613 | 0.5986 | 0.6440 | 0.5935 | 0.9647 | 0.9733 | 0.9447 | 0.9588 | 0.7568 | 0.6954 | 0.6777 | 0.6837 |

Table 10

For the "Congressional Voting" datasets, since this dataset is relatively small and highly structured, it has very high scores across all metrics. Notably, the 70/30 split shows perfect precision (1.0000), which can either be caused by label imbalance or a model predicting only one class correctly. Since in Figure 1, the imbalance is clear, 90 to 130, this is likely the cause. This occurs in multiple split settings (70/30 and 90/10) and is likely an artifact of class imbalance in the test set for those folds. Interestingly, the cross-validation scores are more balanced and realistic, suggesting that stratified k-fold CV provides a more reliable estimate by smoothing out variability from split to split.

For the "Amazon Reviews" dataset, performance is generally much lower. This is especially the case for precision and recall, because this dataset is more complex, larger, and likely noisier (subjective user reviews). We see a gradual improvement in accuracy from 50/50 to 90/10 (from 0.5120 to 0.6400), which may seem counterintuitive but is a consequence of having more training data in larger splits. However, the cross-validation scores, particularly with 10 folds, offer more stability and robustness, showing that cross-validation helps mitigate overfitting by validating on diverse data slices, even if runtime increases.

In terms of the "Airline passenger Satisfaction" dataset, it consistently shows very high performance across all metrics, with small fluctuations between splits and CV folds. This likely indicates that the features are highly predictive and that the data is well-behaved. The results across 50/50, 70/30, and 90/10 are extremely close, which suggests the model isn't very sensitive to split ratio here. Notably, cross-validation further improves generalization, showing consistent gains in precision and F1-score.

The "Student Dropout" dataset is moderately complex and shows moderate performance overall. The results fluctuate slightly with changes in split sizes, but not dramatically. Precision and recall seem to balance out, and again, cross-validation shows consistent scores with slightly better F1-score due to aggregated exposure to all data points across folds. This reinforces that cross-validation is especially useful when datasets aren't massive but contain heterogeneity or class imbalance.

Runtime varied across datasets and evaluation types. Most training runs are completed within 7 minutes. However, cross-validation on the Airline dataset exceeded 2 hours in initial CPU-only trials.

This is directly attributable to the large size of the dataset (tens of thousands of records), the deep architecture used in CV, and the need to retrain the model from scratch in each fold.

After installing PyTorch with CUDA support and switching to GPU acceleration, this runtime dropped significantly to around 2.5 minutes, demonstrating how hardware acceleration is crucial for large datasets and deep models. It's worth highlighting that for the train/test splits, we used a base model with moderate depth, while for the cross-validation experiments, we opted for a deeper, regularized MLP architecture (e.g., with multiple dropout layers and larger hidden layers). This will be discussed more thoroughly in the next section, where we justify and analyze the model design decisions.

## *Influence of Model Architecture on Performance*

In this section, we will analyze how using four different MLP architectures (in depth, width, activation functions, and regularization strategies like dropout affects the resulting performance of the models on our diverse datasets. We break down the observed trends by dataset, analyzing how architectural changes affect accuracy, precision, recall, and F1-score. Firstly, the results are in the tables below.

| Classifier Description | Congressional Voting | | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F1-Score |
| nn.Linear(X.shape[1], 16)<br>nn.ReLU()<br>nn.Linear(16, 8)<br>nn.ReLU()<br>nn.Linear(8, 2) | 0.9848 | 1.0000 | 0.9643 | 0.9818 |
| nn.Linear(X.shape[1], 64)<br>nn.ReLU()<br>nn.Linear(64, 32)<br>nn.ReLU()<br>nn.Linear(32, 16)<br>nn.ReLU()<br>nn.Linear(16, 2) | 0.9848 | 1.0000 | 0.9643 | 0.9818 |
| nn.Linear(X.shape[1], 32)<br>nn.Tanh()<br>nn.Linear(32, 16)<br>nn.Tanh()<br>nn.Linear(16, 2) | 0.9848 | 1.0000 | 0.9643 | 0.9818 |
| nn.Linear(X.shape[1], 128)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(64, 2) | 0.9697 | 0.9643 | 0.9643 | 0.9643 |

Table 11.A

| Classifier Description | Amazon Reviews | | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F1-Score |
| nn.Linear(input_dim, 256)<br>nn.ReLU()<br>nn.Linear(256, output_dim) | 0.4889 | 0.5764 | 0.4730 | 0.4691 |
| nn.Linear(input_dim, 512)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(512, 256)<br>nn.ReLU()<br>nn.Linear(256, output_dim) | 0.5378 | 0.6020 | 0.5293 | 0.5144 |
| nn.Linear(input_dim, 512)<br>nn.ReLU()<br>nn.Linear(512, 256)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(256, 128)<br>nn.ReLU()<br>nn.Linear(128, output_dim) | 0.5200 | 0.5755 | 0.5130 | 0.5013 |
| nn.Linear(input_dim, 1024)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(1024, 512)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(512, 256)<br>nn.ReLU()<br>nn.Linear(256, output_dim) | 0.5689 | 0.6306 | 0.5483 | 0.5350 |

Table 11.B

| Classifier Description | Airplane Passenger Satisfaction | | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F1-Score |
| nn.Linear(input_dim, 64)<br>nn.ReLU()<br>nn.Linear(64, 32)<br>nn.ReLU()<br>nn.Linear(32, output_dim) | 0.9511 | 0.9554 | 0.9309 | 0.9430 |
| nn.Linear(input_dim, 128)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Linear(64, output_dim) | 0.9575 | 0.9675 | 0.9336 | 0.9502 |
| nn.Linear(input_dim, 256)<br>nn.ReLU()<br>nn.Linear(256, 128)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Linear(64, output_dim) | 0.9571 | 0.9570 | 0.9437 | 0.9503 |
| nn.Linear(input_dim, 512)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(512, 256)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(256, 128)<br>nn.ReLU()<br>nn.Linear(128, output_dim) | 0.9630 | 0.9724 | 0.9415 | 0.9567 |

Table 11.C

| Classifier Description | Students' Dropout and Academic Success | | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F1-Score |
| nn.Linear(input_dim, 64)<br>nn.ReLU()<br>nn.Linear(64, output_dim) | 0.7304 | 0.6710 | 0.6677 | 0.6692 |
| nn.Linear(input_dim, 128)<br>nn.ReLU()<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Linear(64, output_dim) | 0.7056 | 0.6361 | 0.6360 | 0.6361 |
| nn.Linear(input_dim, 128)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(64, output_dim) | 0.7613 | 0.6988 | 0.6849 | 0.6903 |
| nn.Linear(input_dim, 256)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(256, 128)<br>nn.ReLU()<br>nn.Dropout(0.3)<br>nn.Linear(128, 64)<br>nn.ReLU()<br>nn.Linear(64, output_dim) | 0.7440 | 0.6796 | 0.6713 | 0.6748 |

Table 11.D

The “Congressional Voting” dataset, being relatively small and well-structured, continued to give strong performance across all metrics with the simpler models. MLP1, MLP2, and MLP3, despite having different depths and activation functions, achieved identical results (accuracy: 0.9848, precision: 1.0000, recall: 0.9643, F1-score: 0.9818), shown in Table 11.A. Our conclusion is that the task is straightforward enough that architectural complexity offers little benefit. Interestingly, the last model (MLP4), which introduced dropout layers, performed slightly worse on all metrics. We believe this is because regularization was unnecessary and may have reduced the model's ability to fully utilize the limited training data. In this context, the simpler models without dropout were better suited to the dataset.

For the “Amazon Reviews” dataset, for which results can be found in Table 11.B, generating results significantly better than a coin flip was very difficult. MLP1 had the lowest accuracy and F1-score, while MLP4, the deepest model with multiple dropout layers, achieved the highest performance (accuracy: 0.5689, F1-score: 0.5350). This shows that model complexity and regularization both play a crucial role when dealing with unstructured and noisy data. Dropout, in particular, appears to have helped generalization by mitigating overfitting.

For the “Airline Passenger” dataset (Table 11.C), all models achieved high scores, reflecting the fact that the classification task is well-defined and the features are informative. MLP1 already performed well, and the performance improved slightly with each more complex model. MLP4 produced the best results (accuracy: 0.9630, F1-score: 0.9567), suggesting that while depth and dropout did help, the gains were incremental. Given the size of the dataset, the deeper model had enough data to learn from without overfitting, and the added regularization further supported generalization. However, the improvements compared to simpler architectures were marginal, indicating that even shallow networks are viable when the dataset is clean and balanced.

For the “Student’s Dropout and Academic Success”, which is more complex than Congressional Voting but smaller and less structured than Amazon Reviews, we saw a more nuanced pattern (review Table 11.D). MLP3, which included two dropout layers, performed the best overall (F1-score: 0.6903). MLP2, which increased depth without applying dropout, had the lowest scores, which we interpret as a sign of overfitting due to the absence of regularization. Interestingly, MLP1, the simplest model, still achieved competitive results. This highlights that simplicity can be effective in moderately complex domains, provided the model is not excessively deep or unregularized.

Our results confirm that the effectiveness of MLP architectures depends on the dataset, and our predictions match those identified in the “Overview of Multi-Layer Perception” section of this report.

# Comparisons Between Classifier Categories

Now that we are finished with the comparison of the models of the same categories, for the last section, we will review how the different categories of classifiers compare against each other. First we will give our predictions, and then we will compare the different models with regard to the metrics defined in the ‘Target Metrics’ section of this report.

## Our Predictions

Given the small size, low feature count, and highly structured nature of the “Congressional Voting” dataset (binary features with little noise), we expect KNN and Random Forest to perform very well. However, we predict that Random Forest will be the best model overall, because it can handle small datasets very efficiently without being sensitive to scaling, while KNN might be slightly more

affected if scaling or imputation isn't perfect. MLP should also perform reasonably well due to the simplicity of the task, but we do not expect it to significantly outperform Random Forest.

For the highly dimensional "Amazon Reviews" dataset, with a very high number of features (10,000) and relatively few instances (750), we expect that Random Forest will perform best here, as it is more robust to high-dimensional and sparse data compared to KNN and MLP. We predict that KNN will perform the worst, as the curse of dimensionality will make distance measurements meaningless. The MLP might perform slightly better than KNN with the right architecture and regularization, but we do not expect it to outperform Random Forest without significant tuning.

Since the "Congressional Voting" dataset is very large (over 100,000 instances) and the features are relatively structured, we predict that MLP will be the best-performing model. MLPs generally excel with large datasets and can learn complex patterns when enough data is available. Random Forest should also perform very well, but we expect the MLP to slightly edge it out if properly tuned. KNN, while possibly achieving good accuracy, will likely suffer in terms of runtime and scalability because of its lazy evaluation approach, making it impractical on this large dataset.

For the "Students' Dropout and Academic Success" dataset, which consists of 4,424 instances with primarily numeric features and a three-class classification target, we predict that Random Forest will perform best. Its natural ability to handle multi-class problems, combined with its robustness to feature scaling and moderate noise, makes it well-suited for this dataset. We also anticipate that the MLP could achieve strong results, although it might be more sensitive to the slight class imbalance and feature variability present in the dataset. Finally, we expect KNN to deliver only moderate performance, as its reliance on distance metrics could be negatively affected by the feature distributions and the need for careful scaling in this particular dataset.

## Final Results

After completing the experiments, we conclude that our predictions were generally accurate. With some key mistakes. This section is entirely based on the results shown in the 'Model Performance' section of this report. Consequently, whenever we refer to results, we refer to the tables in those sections. We will compare the best model with our target metrics, and we will look at some interesting results. Before we begin with the comparisons, it is worth stating that, with a few exceptions, results often only differed by ~1%, not only across splits and different hyperparameters, but also across models, something which we really didn't expect.

### *Best Results Comparison*

For the "Congressional Voting" dataset, we could not get any model to outperform the RF. The extremely simple approach always seemed to get it right for this dataset, even being the final model submitted to the Kaggle competition. The RF managed to achieve 100% accuracy, precision, recall, and F1-score on the 70/30 and 90/10 splits, with the 50/50 splits giving the most reasonable results (97.25% for accuracy and ~97.2% for all other metrics). Although this was possibly the result of overfitting (a hypothesis which we will prove in the next section of this report, 'Interesting Results Comparison'). The MLP and KNN got the worst results overall.

For the "Amazon Reviews" dataset, the Results were generally close between the RF and the MLP, both achieving an average accuracy of ~ 60%, with the RF being slightly better at 62% average. The KNN performed terribly in this context, as high dimensionality is not its strong suit.

In the case of the “Airplane Passenger Satisfaction” dataset, as we expected, MLP achieved the best results here. The deeper MLP model with dropout layers achieved 96.30% accuracy and a very high F1-score of 95.67%, outperforming the RF’s 94.9% accuracy slightly. This confirmed our prediction that the large dataset size and clean structure would favor deep neural networks over traditional ensemble methods.

Lastly, for the “Student’s Academic Success Datasets, again, RF and MLP were quite close. Random Forest achieved a top accuracy of 78.33% and an F1-score of 68.72%, while the best MLP variant (with dropout) achieved 76.13% accuracy and 69.03% F1-score. Interestingly, while Random Forest had slightly higher accuracy, MLP achieved a slightly better F1-score, indicating better balanced classification across all three classes. Overall, both models performed competitively, but depending on the specific metric of interest (accuracy vs balanced performance), either model could be considered "best."

### *Interesting Results Comparison*

Finally, we get more interesting results. Here, despite its known weaknesses with large datasets, KNN achieved up to 97.95% accuracy when trained on 70% of the “Airplane Passenger Satisfaction” dataset, showing that with sufficient instances and proper scaling, KNN can still perform well, even if prediction times remain impractical at scale. Next, we noticed that cross-validation significantly reduced all over-fitting effects. Tables 5 and 10 clearly show that cross-validation provides more reliable and slightly lower performance metrics compared to holdout splits.

In terms of missing values, our results consistently show that for the two datasets, imputing values was always slightly better than treating them as new categories, or removing them. We tested this for the MLP locally, and it didn’t make much of a difference. Lastly, on the Students’ Dropout dataset, deeper MLPs without regularization tended to overfit slightly, especially on 70/30 and 90/10 splits. Models that included dropout layers performed significantly better, demonstrating that regularization is essential when using MLPs on moderately sized, slightly imbalanced datasets.