# Do we need more bikes?

Albert Bargalló I Sales
Michel Misso
Markus Lindgren
William Berger

December 19, 2024

**Abstract**

This paper explores different machine learning models for predicting whether there will be a low or high demand on shared bicycles in Washington DC. The explored models include Logistic Regression, K-nearest neighbors, Random forest, XGBoost and CatBoost. Hyperparameter tuning was done on all the models and after evaluating them on a test-set with recall as focus metric, random forest was deemed the winner. The random forest model achieved a recall score of 64% and overall accuracy of 88%. The Random Forest model was used to predict on the final test set.

## 1 Introduction

To reduce dependence on fossil fuel-based transportation, Capital Bikeshare offers Washington DC, a public bicycle-sharing system as a sustainable alternative. However, on certain occasions, the demand for bicycles exceeds the availability, which in turn leads to people choosing less sustainable modes of transportation. Thus, the District Department of Transportation in Washington seeks to fix this problem by providing more bicycles during times of high demand. This project aims to use a machine-learning approach to provide insight in when it would be beneficial to increase the number of bicycles. By creating multiple models for binary classification and evaluating them against each other, the goal is to use the best model to predict if there is high or low demand for bicycles each hour.

## 2 Data analysis

### 2.1 Exploring features

Through an initial exploratory analysis of the dataset provided, the structure and types of features available for modeling were examined. This analysis involved identifying the numerical and categorical features.

```
"Numerical": ["temp", "dew", "humidity", "precip", "snow",
          "snow_depth", "windspeed", "cloudcover", "visibility"],
"Categorical": ["hour_of_day", "day_of_week", "month", "holiday",
                         "weekday", "summertime"]
```

## 2.2 Trends

As it can be seen in Figure 1, comparing the demand of bikes on different hours it can be derived that the highest demand occurs on the afternoon (15-19h), and that there is also a higher demand during the morning (8-14h) than at night (20-7h). Analyzing for each month, It seems that there is a considerably higher demand of bikes from March to October than from November to February.

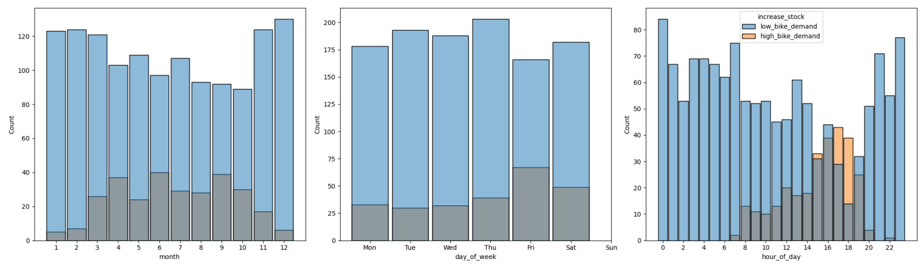

Figure 1: Bike demand plotted against temporal data.

Table 1 highlights a clear difference in high bike demand between weekends and weekdays. The percentage of high bike demand during weekends is 25%, significantly higher compared to 15.14% on weekdays. This indicates that weekends experience a notable increase in bike usage. Table 2 shows that high bike demand during non-holidays (18.03%) is only slightly higher than during holidays (16.98%), indicating that the "holiday" variable had minimal impact on bike demand.

Table 1: High demand by day of week.

| **Name** | **Description** |
|---|---|
| Weekends: | 25% |
| Weekdays: | 15.14% |

Table 2: High demand by holiday.

| **Name** | **Description** |
|---|---|
| Non-holiday: | 18.03% |
| Holiday: | 16.98% |

The histograms of temperature and windspeed in Figure 2 provide insights into the relationship between these variables and bike demand. The windspeed histogram shows a similar distribution pattern for both low and high bike demand, with a larger volume of data corresponding to low bike demand. This indicates that windspeed does not strongly differentiate between the two demand categories, suggesting it may not be a key predictor for high bike demand. In contrast, the temperature histogram reveals a distinct trend. Higher temperatures are clearly associated with increased bike demand, as evidenced by the greater proportion of high bike demand at elevated temperatures. Conversely, lower temperatures predominantly correspond to low bike demand. This trend underscores the significant role that temperature plays in influencing bike demand, making it a critical feature in predicting demand patterns. These observations reinforce the importance of focusing on variables like temperature that exhibit strong predictive signals, while less impactful variables like windspeed and holiday may contribute less to the model's performance and could be removed.

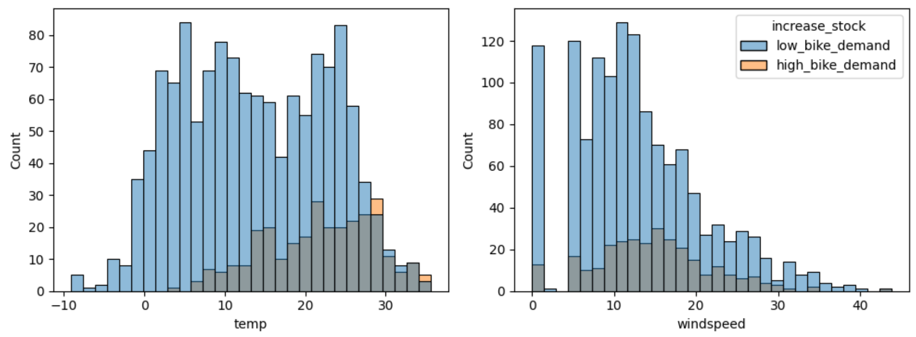

Figure 2: Bike demand plotted against weather data.

## 2.3 Feature selection

Based on the data analysis above, a new dataset was created with the most important features. A correlation matrix and plots similar to the figures above were created to help with this process.

First, the column *holiday* was removed based on the minimal information it contained. The column *snow* was also dropped since the entire column only contained 0:s which gives us no additional information on the demand of bikes. Secondly, the *snowdepth* feature was converted to a binary feature *snow*, 1 if there was any snow and 0 otherwise, as it seems to be a significant difference in the demand when there is snow and when there isn't, but not between the amount of snow. *Precip* was also converted to a binary feature *rain*, since the amount rainy days with high bicycle demand was not many. In addition, it was decided to keep the *summertime* feature since there is a clear difference in the demand between summer/winter. Lastly, a new binary feature called *day* was created, being 1 if *hour_ of_ day* was between 7 and 20h, as it could be useful to represent the difference on demand by *hour_ of_ day* seen on Figure 1.

Furthermore, the numerical features in the dataset were normalized through *Z-score scaling* to ensure that they were centered around 0 with a standard deviation of 1. The reason for this was to help the model converge quickly and possibly give more accurate predictions.[1] The normalization step is particularly important for logistic regression and k-NN since they rely on distance measurement to learn. If the feature ranges are very different, the calculations can be dominated by features with larger ranges, leading to biased or suboptimal results.[1] We also decided not to one-hot encode any categorical values since *hour_ of_ day*, *month*, and *day_ of_ week* all have a natural ordering. One-hot encoding also increases the dimensionality of the data, and in our case a lot since there are many categories in each feature. This would have led to slower training and very sparse data since most of the rows would be 0 in these columns.[2]

## 3 Model development

### 3.1 Logistic Regression

Logistic regression is one of the main basic models used for binary classification. In the case involved in this project, the aim is to predict if there will be high or low availability of bikes during a specific hour, that is, aiming to predict that a given observation belongs to Y = 1 (Y being the target variable) based on some input features (x). Given a linear classifier: $f(x;\theta) = sign(x^T\theta)$. To train the model, the idea is to optimize the parameters ($\theta$) by minimizing the logistic loss function ($L$) (another approach would be to maximize the likelihood): $L(x,y;\theta) = \ln[1 + exp(-yx^T\theta)]$ After obtaining the probability $\hat{p} = f(x;\theta)$, the prediction is converted to a binary class label using a threshold $t$ (commonly 0.5). The logistic regression algorithm was implemented using the sk-learn library in python. Through grid search performed with 5 cross validation folds, the optimal hyperparameters for the model where found. They are detailed below.

```
Hyperparameters {"C": 100, "penalty": "l1", "solver": "liblinear"}
```

### 3.2 K-Nearest Neighbors

K-Nearest Neighbors, is a non-parametric supervised learning classifier, and uses proximity to make classifications. A class label is assigned using a plurality system. To decide what class a given point should be assigned we need some way to measure the distance between that point and its k neighbors. This distance creates decision boundaries, which can be visualized using Voronoi diagrams.

$$\text{Minkowski Distance} = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}} \tag{1}$$

Using the Minkowski distance as a generalized distance formula, we can create the two most common distance formulas used in k-nearest neighbors. When p = 2 we have the *euclidian* distance, and when p = 1, we have the *manhattan* distance [3]. The k-value determines how many neighbors are checked when the model classifies a new point. The k-nearest neighbor algorithm was implemented using the scikit-learn library in python. A grid search cross validation with 5 folds was used to find the optimal hyper-parameters, and below, we can see what these optimal hyper-parameters are (note that the other hyperparameters that scitkit-learn's KNeighborsClassifier support were left to default values).

```
Hyperparameters { "algorithm": ball_tree, "metric": manhattan,
                  "n_neighbors": 23, "weights": uniform }
```

## 3.3 Random Forest

Random forest is an ensemble method closely related to bootstrap aggregation (bagging). Bagging is a technique that averages over multiple low-bias high-variance models to reduce variance. There are two things that separate bagging and random forest. The first thing is that bagging can be used with any base model, while random forest assumes that the base models are classification or regression trees. The second thing is the randomness that the random forest method injects when constructing the trees. Instead of considering all features $x_1, ..., x_p$ as splitting variables, random forest picks $q \leq p$ random splitting variables and only considers these. This random selection of $q$ is made at each splitting point in each of the ensemble trees $B$. The random selection of features will make the $B$ trees less correlated, and thus the average over the trees predictions can result in even more variance reduction than bagging.[4] The random forest algorithm was implemented with the scikit-learn library in python. The models best hyperparameters were found by doing a grid search with $K - folds = 5$. The best parameters are shown below.

```
Hyperparameters { "bootstrap": True, "max_depth": 20,
"max_features": sqrt, "n_estimators": 50, "min_sample_leaf": 1,
"min_sample_split": 2 }
```

## 3.4 Boosting

Boosting is a machine learning method in which multiple simple models, like small decision trees, are combined to make a stronger model. The simple model is an algorithm that generates classifiers with an error rate slightly better than random guessing (e.g., less than 0.5 in binary classification). In contrast, a strong learner with sufficient training data can produce classifiers with a very low error probability. The algorithm could be written as detailed below.

$$F(x) = \sum_{t=1}^{T} \alpha_t h_t(x), \tag{2}$$

where T = Total number of simple learners, $\alpha_t$ = Weight (or importance) of the $t$-th simple learner and $h_t(x)$ = The $t$-th simple learner.

### 3.4.1 XGBoost

XGBoost (Extreme Gradient Boosting) was chosen as one of the boosting algorithms for this problem because of its ability to handle structured datasets effectively and model complex non-linear relationships between variables. The dataset includes a mix of temporal features, such as hour_of_day and day_of_week, and weather-related variables like temp, humidity, and rain. XGBoost is well suited for datasets of this nature, as it captures interactions between these features without requiring extensive preprocessing.

```
Hyperparameters { "gamma": 0.1, "max_depth": 10, "reg_alpha": 1,
      "learning_rate": 0.4, "n_estimators": 20, "reg_lambda": 0 }
```

#### 3.4.2 CatBoost

Categorical Boosting was also considered for this problem due to its ability to handle categorical features directly. Unlike XGBoost, CatBoost processes categorical data natively and employs ordered boosting, which mitigates target leakage and enhances the robustness of the model. This is especially useful in this scenario where the dataset is quite small and overfitting can be a concern. After performing grid search over 5 cross validation folds, the optimal hyperparameters found where the ones stated below.

```
Hyperparameters { "depth": 6, "iterations": 1000,
                 "learning_rate": 0.01 }
```

## 4 Model Selection & Results

All the results are obtained from models trained on 80% of the dataset, and then evaluted on the remaining 20%. Random state/seed = 0 was used throughout the code, both for the dataset splits, as well as for the models that accepts a random state as an argument.

Table 3: Performance metrics for all of the models considered.

| Model | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| Logistic Regression | 0.8688 | 0.6538 | 0.5862 | 0.6182 |
| KNN | 0.8656 | 0.6531 | 0.5517 | 0.5981 |
| Random Forest | **0.8812** | 0.6852 | **0.6379** | **0.6607** |
| XGBoost | 0.8656 | 0.6415 | 0.5862 | 0.6126 |
| CatBoost | 0.8781 | **0.7021** | 0.5690 | 0.6286 |
| Naive Classifier | 0.8187 | 0 | 0 | 0 |

Looking at the ROC Curves, and the respective areas under the curves, it can be seen that *CatBoost* performs the best. But looking at all other metrics; accuracy, precision, recall, and f1-score, as outlined in Table 3, Random Forest performs the best on all of them, except precision where CatBoost is the best. Due to the nature of the problem, recall was deemed to be important to prioritize.In this case, a higher recall value implies that the model does not wrongly predict the negative class, which is low bike demand in this case. In a business perspective, it is bad to predict low demand when in the reality, the demand is high, as customers will chose other modes of transportation. Seeing as Random Forest displays the highest results on the most important metrics presented above, and a quite significantly higher recall than the other models, this model will be used in production.

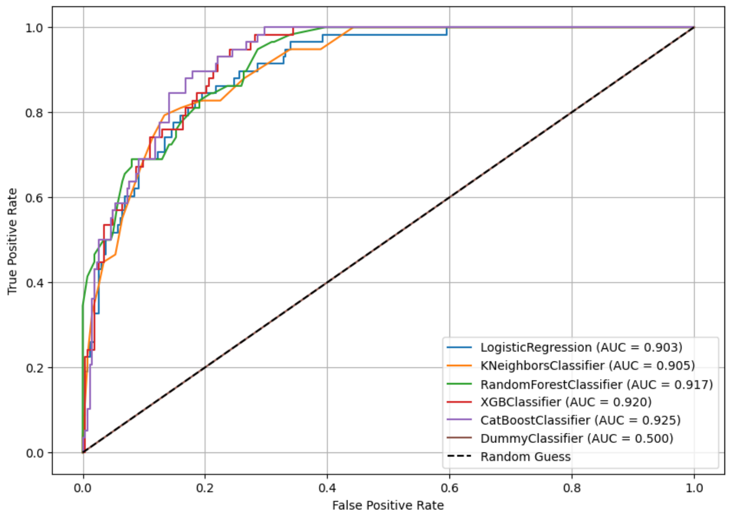

Figure 3: ROC curves of all of the models considered.

## 5 Conclusions

This study demonstrates the effectiveness of machine learning models in predicting whether there will be high or low demand of bikes for Washington DC's Capital Bikeshare system. While all the contemplated models outperform the naive classifier, Random Forest stands out from the rest by achieving the best performance in most metrics, including accuracy, precision, and F1-Score. With an AUC of 0.917, it is only slightly behind CatBoost's top score of 0.925. However, Random Forest's recall is 12% higher than CatBoost's, representing a more optimal trade-off between accuracy and responsiveness to high-demand conditions. In this project recall is particularly critical, as it aligns with the goal of minimizing instances where high demand is incorrectly predicted as low, reducing the risk of bike shortages during peak demand.

In conclusion, the results evidence that Random Forest is the most reliable choice for deployment among all the models studied. This model provides a robust, data-driven solution for optimizing bike availability and promoting the use of sustainable transportation in Washington DC.

## References

[1] G. Developers, “Normalization,” https://developers.google.com/machine-learning/crash-course/numerical-data/normalization, 2024, accessed on December 4, 2024.

[2] GeeksforGeeks, “One hot encoding,” https://www.geeksforgeeks.org/ml-one-hot-encoding/, accessed on December 4, 2024.

[3] IBM, “What is the k-nearest neighbors (knn) algorithm?” 2024, accessed: 2024-12-08. [Online]. Available: https://www.ibm.com/topics/knn

[4] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning - A First Course for Engineers and Scientists.* Cambridge University Press, 2022. [Online]. Available: https://smlbook.org

# A Appendix

## A.1 dataloader.py

This library was created and used for loading and splitting the dataset.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

def load_and_split_data(filepath, target_column, class_zero,
    test_size=0.2, random_state=0, cat_features=None):
    """
    Loads data from a CSV file, processes it, and returns train-
    test splits.

    Parameters:
        filepath (str): Path to the CSV file.
        target_column (str): The name of the target column to be
    predicted.
        class_zero (str): The name of the class to be used as the
    reference class (0).
        convert_cat_target (bool): Whether to convert the target
    column to binary.
        test_size (float): Fraction of the data to use as test set.
        random_state (int): Random seed for reproducibility.
        cat_features (list): List of categorical features to be
    converted to category type.

    Returns:
        tuple: X_train, X_test, y_train, y_test
    """

    df = pd.read_csv(filepath)

    # Assign 0 to the class_zero and 1 to the other class in the
    target column
    df[target_column] = np.where(df[target_column] == class_zero,
    0, 1)

    if cat_features:
        for feature in cat_features:
            df[feature] = df[feature].astype('category')

    # Split into features and target
    X = df.copy()
    y = X.pop(target_column)

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=test_size, random_state=random_state)

    return X_train, X_test, y_train, y_test
```

## A.2 feature_eng.py

This library was created and used for preprocessing the dataset.

```python
import pandas as pd


def preprocess(path_in, path_out, name_out):
    """
    Preprocess the dataset and saves it as a .csv file.

    Args:
        path_in (str): path to the dataset
        path_out (str): path to save the preprocessed dataset
        name_out (str): name for the preprocessed dataset
    """

    # Load the dataset
    data = pd.read_csv(path_in)

    # Define the numerical features
    num_features = ['temp', 'dew', 'humidity', 'windspeed', '
    cloudcover', 'visibility']

    # Drop holiday and snow columns
    data = data.drop(columns=['holiday', 'snow'])

    # Add a binary feature called "day" where 1 means "hour_of_day"
     is between 7 and 20, and 0 otherwise
    data['day'] = ((data['hour_of_day'] >= 7) & (data['hour_of_day'
    ] <= 20)).astype(int)

    # Encode "snowdepth" as a binary feature where 1 means if there
     is snow and 0 otherwise
    data['snowdepth'] = (data['snowdepth'] > 0).astype(int)

    # Add a binary feature called "rain" where 1 means if "precip"
    is greater than 0, and 0 otherwise
    data['rain'] = (data['precip'] > 0).astype(int)

    # Drop "precip" column
    data = data.drop(columns=['precip'])

    # Normalize the numerical features
    for feature in num_features:
        data[feature] = (data[feature] - data[feature].mean()) /
    data[feature].std()

    # Save the preprocessed dataset as csv
    data.to_csv(path_out + name_out, index=False)
```

## A.3 utils.py

This library was created and used for developing and training the models.

```python

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, confusion_matrix, roc_curve
from sklearn.model_selection import GridSearchCV
import json
import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd




def find_optimal_hyperparameters(model, param_grid, X_train, y_train, cv=5, scoring='accuracy', n_jobs=-1, save_dir="", save_file='knn_best_params.json', extra_args={}, verbose_training=False):

    """
    Find the optimal hyperparameters for a model using GridSearchCV

    Parameters:
        model: The model class
        param_grid (dict): The hyperparameters to search over
        X_train (pd.DataFrame): The training data
        y_train (pd.Series): The training labels
        cv (int): The number of cross-validation folds
        scoring (str): The scoring metric
        n_jobs (int): The number of jobs to run in parallel
        save_dir (str): The directory to save the best parameters
        save_file (str): The file to save the best parameters
        extra_args (dict): Extra arguments to pass to the model
        verbose_training (bool): Whether to print the training progress

    Returns:
        dict: The best hyperparameters found
    """

    model = model(**extra_args)
    gs_cv = GridSearchCV(model, param_grid, cv=cv, scoring=scoring, n_jobs=n_jobs)

    if 'CatBoostClassifier' in str(model):
        gs_cv.fit(X_train, y_train, verbose=verbose_training)
    else:
        gs_cv.fit(X_train, y_train)

    best_params = gs_cv.best_params_
    print("Best parameters found: ", best_params)

    if extra_args:
        best_params.update(extra_args)

    if save_dir:
        print("Saving best parameters to '{}'".format(os.path.join(save_dir, save_file).replace('\\', '/').strip()))
        with open(os.path.join(save_dir, save_file), 'w') as f:
            json.dump(best_params, f)

    return best_params


def load_model_from_json(model, json_file):
    """
    Load a model from a json file

    Parameters:
        model: The model class to load
        json_file (str): The path to the json file

    Returns:
        model: The model loaded from the json file
    """

    with open(json_file, 'r') as f:
        params = json.load(f)

    model = model(**params)

    return model

def plot_roc_curves(results):
    """
    Plot the ROC curves for the models

    Parameters:
        results (dict): The results from fit_and_evaluate_multiple

    Preconditions:
        - results contains the fpr and tpr for each model, as well as the roc_auc
    """
    plt.figure(figsize=(10, 7))
    for model_name, metrics in results.items():
        plt.plot(metrics["fpr"], metrics["tpr"], label=f"{model_name} (AUC = {metrics['roc_auc']:.3f})")

    plt.plot([0, 1], [0, 1], 'k--', label="Random Guess")
    plt.title("ROC Curves")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.legend(loc="lower right")
    plt.grid()
    plt.show()

def fit_and_evaluate_multiple(models, X_train, y_train, X_test, y_test, verbose=False, verbose_training=False, float_precision=4):
    """
    Fits multiple models on the given data and evaluates them on the testing data.

    Parameters:
        models (list): The models
        X_train (pd.DataFrame): The training data
        y_train (pd.Series): The training labels
        X_test (pd.DataFrame): The testing data
        y_test (pd.Series): The testing labels
        verbose (bool): Whether to print the results
        verbose_training (bool): Whether to print the training progress
        float_precision (int): The number of decimal places to print

    Returns:
        dict: The accuracy, precision, recall, F1, ROC AUC, and confusion matrix for each model
    """

    results = {}
    for model in models:
        results[model.__class__.__name__] = fit_and_evaluate(model, X_train, y_train, X_test, y_test, verbose, verbose_training, float_precision)
    return results

def fit_and_evaluate(model, X_train, y_train, X_test, y_test, verbose=False, verbose_training=False, float_precision=4):
    """
    Fits a model on the given data and evaluates it on the testing data.

    Parameters:
        model: The model
        X_train (pd.DataFrame): The training data
        y_train (pd.Series): The training labels
        X_test (pd.DataFrame): The testing data
        y_test (pd.Series): The testing labels
        verbose (bool): Whether to print the results
        verbose_training (bool): Whether to print the training progress
        float_precision (int): The number of decimal places to print

    Returns:
        dict: The accuracy, precision, recall, F1, ROC AUC, confusion matrix, fpr, tpr
    """

    if 'CatBoostClassifier' in str(model):
        model.fit(X_train, y_train, verbose=verbose_training)
    else:
        model.fit(X_train, y_train)
    return evaluate(model, X_test, y_test, verbose, float_precision)

```

```python
def evaluate(model, X_test, y_test, verbose=False, float_precision=4):
    """
    Evaluates a model on the given data and returns the accuracy, precision, recall, F1, ROC AUC, and confusion matrix, fpr, tpr as a dictionary.

    Parameters:
        model: The model
        X_test (pd.DataFrame): The testing data
        y_test (pd.Series): The testing labels
        verbose (bool): Whether to print the results
        float_precision (int): The number of decimal places to print
    Returns:
        dict: The accuracy, precision, recall, F1, ROC AUC, confusion matrix, fpr, tpr
    """

    y_pred = model.predict(X_test)
    y_pred_prob = model.predict_proba(X_test)[:, 1]

    acc = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, zero_division=0)
    recall = recall_score(y_test, y_pred, zero_division=0)
    f1 = f1_score(y_test, y_pred, zero_division=0)
    roc_auc = roc_auc_score(y_test, y_pred_prob)
    cm = confusion_matrix(y_test, y_pred)
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob)

    if verbose:
        print(f"Evaluating {model.__class__.__name__}")
        print(f"Accuracy: {acc:.{float_precision}f}")
        print(f"Precision: {precision:.{float_precision}f}")
        print(f"Recall: {recall:.{float_precision}f}")
        print(f"F1: {f1:.{float_precision}f}")
        print(f"ROC AUC: {roc_auc:.{float_precision}f}")
        print(f"Confusion Matrix: \n{cm}")
        print()

    results_dict = {
        "accuracy": acc,
        "precision": precision,
        "recall": recall,
        "f1": f1,
        "roc_auc": roc_auc,
        "confusion_matrix": cm,
        "fpr": fpr,
        "tpr": tpr,
    }

    return results_dict


def fit_and_save_predictions(model, training_data, X_eval, target_column, class_zero):
    """
    Fits a model on the given data and saves the predictions on the evaluation data.

    Parameters:
        model: The model
        training_data (csv): Path to the training data
        X_eval (csv): Path to the evaluation data
        target_column (str): The name of the target column to be predicted
        class_zero (str): The name of the class to be used as the reference class (0)
    """

    # Load training data
    training_data = pd.read_csv(training_data)

    # Assign 0 to the class_zero and 1 to the other class in the target column
    training_data[target_column] = np.where(training_data[target_column] == class_zero, 0, 1)

    # Split training data into features and target
    X_train = training_data.copy()
    y_train = X_train.pop(target_column)

    # Load evaluation data
    X_eval = pd.read_csv(X_eval)

    # Fit the model on the training data
    model.fit(X_train, y_train)

    # Compute the predictions on the evaluation data
    y_pred = model.predict(X_eval)

    # Reshape the predictions to a single row
    y_pred_row = np.reshape(y_pred, (1, -1))

    # Save the predictions to a CSV file
    y_pred_df = pd.DataFrame(y_pred_row)
    y_pred_df.to_csv("data/final_predictions.csv", header=False, index=False)
```