HAPPINESS PROGRESSION BY REGRESSION

JOSEPH DU TOIT, OSCAR J. ESCOBAR, AND HENRY FETZER

ABSTRACT. With data from the World Bank and World Happiness Report, we analyze the influence of various factors and forces on human happiness. We fit machine-learning models that accurately predict happiness scores and present an information-theoretic method to isolate features that correspond to happiness independently of GDP. Our research offers data-driven insight into how organizations can improve the world's standard of living. See our GitHub repository.

1. RESEARCH QUESTION AND OVERVIEW OF THE DATA

In 2011, the United Nations (UN) passed a resolution entitled, "Happiness: towards a Holistic Approach to Development" in which they claim, that GDP "was not designed to—and [does] not reflect adequately—the happiness and well-being of people" [Ass11]. In this project, we seek to understand how using happiness as a proxy for quality of life may change political and economic priorities from when GDP is used. Specifically, we ask:
1. What is the relationship between a country's happiness and socioeconomic status, and can we predict known happiness data?
2. Can we predict happiness scores for countries where we have no happiness data?
3. What factors influence happiness independently of GDP?
Our research joins several studies using similar methods to approach these questions such as [Zha23], [Eas15], and [ES09]. We hope that finding meaningful answers can elucidate the difference between using happiness and GDP as a developmental objective and inform decisions to improve global happiness.

Since happiness is not directly quantifiable and less importance has been placed on measuring it than GDP, obtaining good data is difficult. The happiness data we use for this project is from the 2022 issue of "The World Happiness Report," published by Gallup, Oxford University, and the UN [HLS+22]. "The World Happiness Report" is the most, reputable study on this topic. Participants from 143 countries are asked to imagine a ladder with 0 being the worst possible life and 10 being the best possible life and to rate their own life on that ladder. A significant limitation of this data is that it is ordinal data, but we will treat it as ratio data, so the scaling in our results is incorrect. We make the simplifying assumption that the data approximates the true distribution well enough to apply analysis. Another problem is that 123 countries are not included, inspiring our interest in predicting data for missing countries.

The social, economic, and political features we use are from the 2022 World Bank Development Indicators —"the most current and accurate global development data available, and includes national, regional and global estimates" [Ban].

Date: December 13, 2024.

This dataset contains 1496 features for 266 countries with both standard metrics and interesting features as specific as the number of firms with female leaders and the number of endangered fish species. However, many entries of this dataset are empty and many features are tightly correlated, which we address with our data-cleaning and feature-selection methods.

2. Data Cleaning / Feature Engineering

Our data pipeline was designed to be dynamic and abstract so that we could easily add new features and iterate experiments. The greatest challenge was handling large empty portions of the feature dataset. We developed a pipeline where all features meeting a completeness criterion are pulled from the online World Bank database into a dataset. Next, we impute missing feature data using K-Nearest-Neighbors (KNN). We decided to use KNN since we found that we had 136 features with no missing values, and we could use the similarity information in these features to estimate missing data better than simply filling them with the feature mean, median, or mode. We also avoided linear imputation to preserve the linear independence of features. For our analysis, we decided to use 390 features that were at least ninety percent complete to have more diverse features while mitigating colinearity due to imputation.

The socioeconomic feature data contains several correlated components, so we implemented methods to identify and engineer the most important features. First, we tried selecting features based on SciPy's `mutual_info_regression` method that estimates mutual information using KNN as introduced by Kraskov et. al. [KSG04]. We selected the 50 features with the highest mutual information with happiness scores to use for regression. Second, we used principal component analysis (PCA) to engineer new features that best express the signal in the data. Using PCA, we captured ninety-five percent of the original 390 features' variance in only 43 new features (see Figs 4 and 5 in the Appendix).

3. Data Visualization and Basic Analysis

To give intuition about the happiness data, we discuss a few summary statistics. The maximum reported happiness score is 7.821 in Finland while the minimum is 2.4038 in Afghanistan. The data has a mean of 5.559 and a standard deviation of 1.088. Refer to 2 for a visualization of the geographic happiness distribution.

We are most interested in the relationship between happiness and GDP per capita as demonstrated in Fig 1. We fit the least-squares linear and log models to the data. The linear model had a mean-squared error of 0.531 and a R^2 value of 0.5483. The log model did better with a mean-squared error of 0.442 and a R^2 value of 0.624. The weak linear correlation was surprising at first, but suggests that the correspondence of GDP per capita with happiness tapers off as GDP per capita increases, so other features may have more predictive power. Fig 1 shows more error in countries with low GDP per capita and happiness scores, which may result from varying scales in the subjective survey responses.

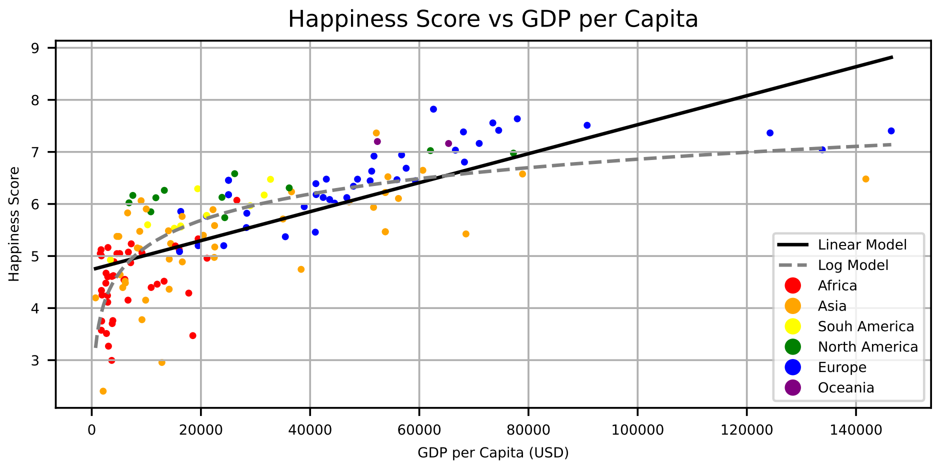

FIGURE 1. Happiness scores against GDP per capita with least-squares linear and log fits. Happiness and GDP per capita are positively correlated, but not strongly.

4. LEARNING ALGORITHMS AND IN-DEPTH ANALYSIS

4.1. **Happiness Prediction with Regression.** First, we attempted to predict happiness scores using all available features. This is a regression problem, so we did not use logistic regression or classification trees. With Bayesian optimization for hyper-parameter tuning (see Appendix B for more info), we trained random forest (RF), XGBoost (see Fig 3), and Linear regressors. We validated our models with 10-fold cross-validation, except for the RF where we employed bootstrapping and Out-of-Bag (OOB) samples. The results are displayed in Table 1. We set our hyper-parameter search for our RF to maximize OOB score while minimizing MSE, so that the RF avoids overfitting. We expected the RF to generalize well due to bagging, and as reported, the RF was the best regressor. The happiness predictions for RF regression can be seen in Figure 2.

Regressor	T-MSE	T-R^2 score	V-MSE	V-R^2 score
RF	0.1056	0.9101	0.3296	0.7195
XGB	0.4180	0.6442	0.5968	0.4433
Linear	0.9944	0.1536	N/A	N/A

TABLE 1. A comparison of the different regression algorithms used to predict happiness scores. T is for the training and V is for the validation.

We also predicted happiness scores for 123 countries excluded from the 2022 World Happiness report using our best performing RF regressor. Although we have no data to measure the model's performance on these countries, the predictions seem to align with expectations as seen in Figure 2 .

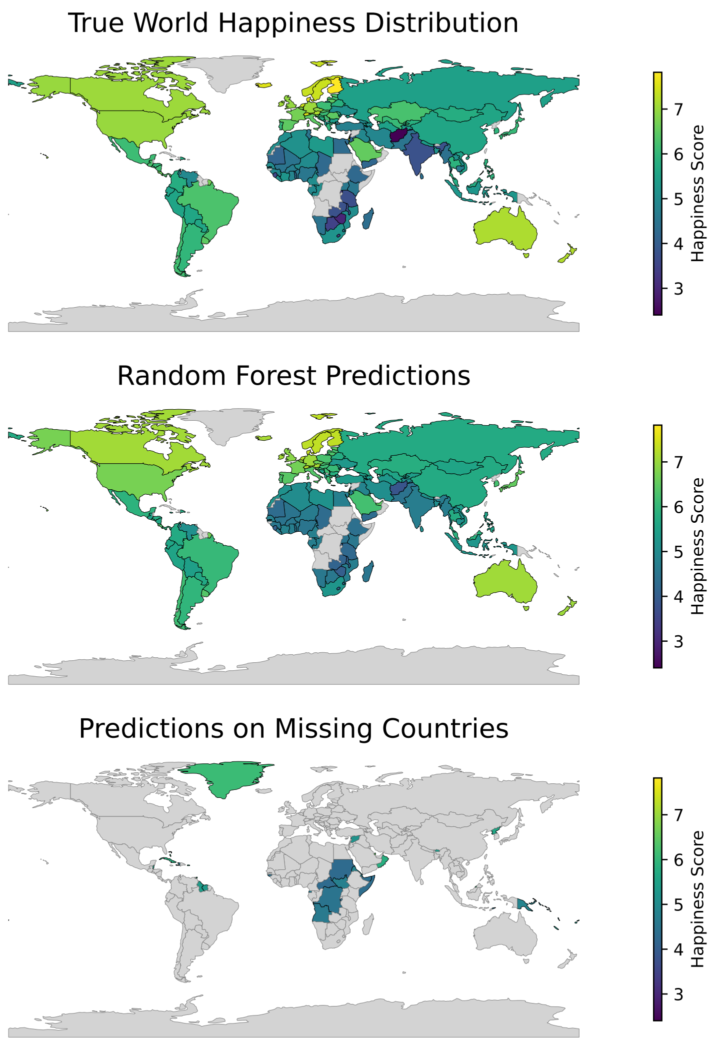
True World Happiness Distribution
Random Forest Predictions
Predictions on Missing Countries
Happiness Score
3
4
5
6
7

FIGURE 2. True happiness scores from the 2022 World Happinesss Report, predicted scores by our RF model, and predictions by our RF model on countries not included in the 2022 World Happiness Report. Countries with data omitted are left grey.

4.2. **Identifying Correlated Features Independent of GDP.** In our last question, we asked what features influence happiness independent of GDP. Essentially, we are trying to find features that share information with happiness but do not share information with GDP. We proceed by considering the mutual information $I(\cdot,\cdot)$ of the distributions for happiness H, GDP G, and each feature F_i. Since mutual information is influenced by scale, we normalize the data. Then, we search for a distribution D^* satisfying

$$D^* = \underset{D}{\operatorname{argmax}}\, I(H,D) - \alpha I(G,D)$$

where α is a tunable parameter weighting the difference between happiness and GDP. We estimate mutual information using KNN with SciPy as described previously. Next, for each feature F_i we compute $I(D^*, F_i)$ to find the feature with a distribution most similar to D.

The results of this experiment were very insightful. For a low weight on differentiation ($\alpha = 0.01$), the distribution D^* had mutual information 3.045 with H. As a reference, the theoretical mutual information maximum of H with itself is $I(H,H) = 3.72$. The top ten most correlated features to this D^* were measurements of GDP per capita, voice/accountability (freedom of speech and representation in government), and child/infant life expectancy. This corroborates the claim that GDP per capita is a good estimator of happiness. For a higher weight ($\alpha = 0.75$), D^* had much lower mutual information with H, $I(H,D^*) = 0.44$, than the previous score suggesting that this minimizer was much less correlated with H. However, there were interesting changes in the top ten features. The new features included voice/accountability metrics, female population and life expectancy, people using safe water sources in cities, birth rate, and political stability/absence of violence and terrorism. A notable implication of these results is that treatment of women may be one of the most important features correlated with happiness outside of GDP per capita.

5. Ethical Implications and Conclusions

Although this investigation has great potential to improve global well-being, we must also consider some of its ethical implications. First, as remarked previously, no proxy metric for standard of living will be comprehensive. Only considering happiness, GDP, or any other proxy for policy development makes the policy susceptible to ignoring other important factors. For example, number of internet users and number of internet subscriptions are in the top 50 features correlated with happiness while total greenhouse emissions is one of the features least correlated with happiness. A government initiative to get everyone cell phones by diverting funds set aside for environmental protection would not be prudent and could cause serious public health concerns. Second, from a malicious perspective, these findings could be used to target factors that maximize suffering in another country. Unfortunately, with conflict rampant in many parts of the world, an aggressor could identify that increasing child mortalities in an opposing country could have much more influence on the country's suffering than the number of imported services and change strategy to attacking civilian structures rather than embargo ports. Third, companies could use our findings to identify products people need most to be happy, and use this knowledge to exploit them through price gouging or creating monopolies. Finally, reducing people's lives to a single "happiness score" replaces their legitimate human experience with an unfeeling number. We recognize that everyone has unique challenges in life that this research cannot model. We encourage the reader to do their part to make the world a happier place.

APPENDIX A. SUPPLEMENTAL GRAPHS

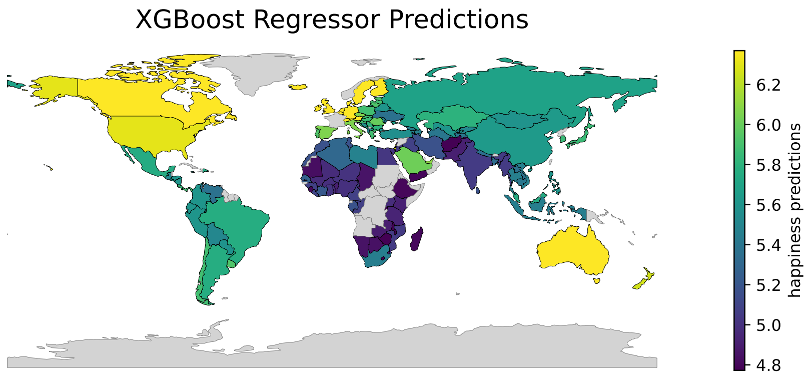

FIGURE 3. The predicted happiness scores using all the features but implemented with XGB regression. Compare to Fig 2.

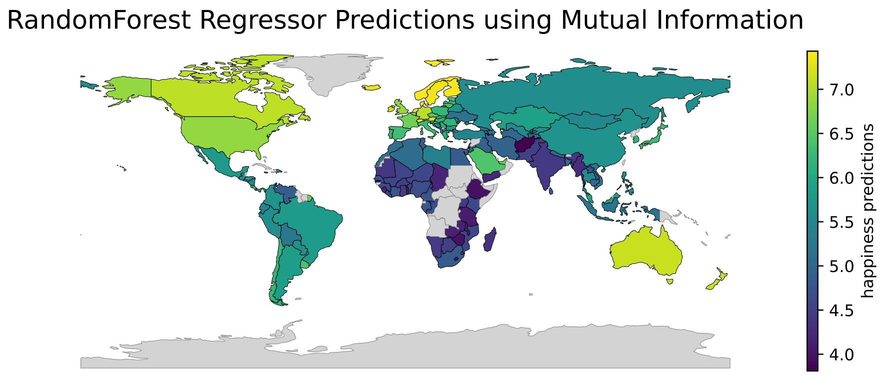

FIGURE 4. The predicted happiness scores using the best 50 features and 6-nearest neighbors. Compare to Figures 2 and 5.

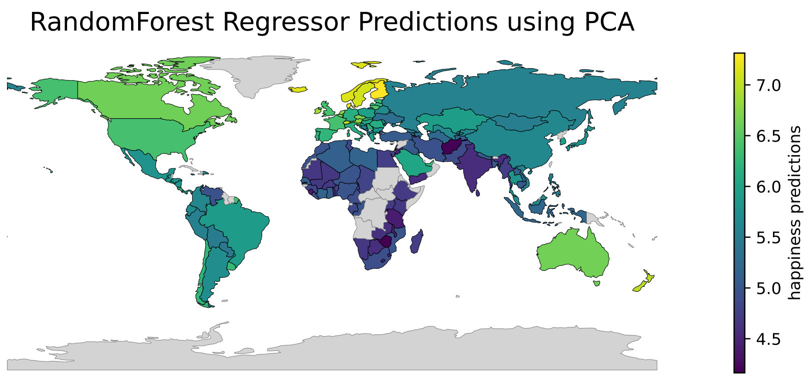

FIGURE 5. The predicted happiness scores using the best 43 principal components that give 95% variance of the original 390 features data. Compare to Figures 4 and 2.

APPENDIX B. EQUATIONS

Specifically, we have that the *mutual information* for two continuous random variables X, Y is

$$I(X,Y) = \int_{\text{supp}(X)} \int_{\text{supp}(Y)} P_{X,Y}(x,y) \log\left(\frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)}\right), \tag{1}$$

where $P_{X,Y}$ is the joint pdf and P_X, P_Y are the respective pdfs for X and Y.

In Bayesian optimization, the goal is to minimize some objective function $f(\mathbf{x})$:

$$\mathbf{x}^* \in \underset{\mathbf{x}\in\mathcal{X}}{\text{argmin}} f(\mathbf{x}),$$

where $\mathcal{X}$ is the domain of the hyperparameter. Specifically, the optimizer we used is the Tree-structure Parzen Estimator (i.e. KDEs) sampler (TPEs) from optuna. TPES uses the TPE algorithm to search the hyperparameter space to find $P(y = f(\mathbf{x})|\mathbf{x}, \mathbb{D}) \propto P(\mathbf{x}|y, \mathbb{D})P(y)$, where $\mathbb{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ is the dataset, by slitting $P(\mathbf{x}|y, \mathbb{D})$ into two probabilities by observing the value of the objective function f. The two probabilities are $P(\mathbf{x}|\mathbb{D}), y \leq y^*$ and$P(\mathbf{x}|\mathbb{D}), y \geq y^*$. The former are for the density values of the objective function that are less than some computed top-quantile values of the observed values y while the latter are density values that are bigger than y^*. For more information, see [Wat23] and [BBBK11].

REFERENCES

[Ass11] UN. General Assembly. Happiness : towards a holistic approach to development : resolution /. *United Nations Digital Library (65th sess. : 2010-2011)*, 2011.

[Ban] World Bank. World development indicators. Accessed: [12/9/2024].

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[Eas15] Richard A Easterlin. Happiness and economic growth: The evidence. 2015.

[ES09] Richard Easterlin and Onnicha Sawangfa. Happiness and economic growth: Does the cross section predict time trends? evidence from developing countries. *International Differences in Well-Being*, 03 2009.

[HLS$^+$22] John F. Helliwell, Richard Layard, Jeffrey D. Sachs, Jan-Emmanuel De Neve, Lara B. Aknin, and Shun Wang. World happiness report 2022, 2022. Accessed from: https://worldhappiness.report/ed/2022/.

[KSG04] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Phys. Rev. E*, 69:066138, Jun 2004.

[Wat23] Shuhei Watanabe. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance, 2023.

[Zha23] Yifei Zhang. Analyze and predict the 2022 world happiness report based on the past year039;s dataset. *Journal of Computer Science*, 19(4):483–492, Mar 2023.

APPENDIX C. CODE

final_nb

December 12, 2024

1 Import Necessary Libraries

```
[ ]: from matplotlib.lines import Line2D
from numpy.typing import ArrayLike, NDArray
from scipy.optimize import minimize
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.feature_selection import mutual_info_regression, SelectKBest
from sklearn.impute import KNNImputer
from sklearn.linear_model import Lasso, LinearRegression, Ridge
from sklearn.metrics import make_scorer, mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score, cross_validate,␣
 ↪GridSearchCV, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from tqdm.auto import tqdm
from typing import Dict, List, Optional, Tuple
from xgboost import XGBClassifier, XGBRegressor


import joblib
import geopandas as gpd
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
import pandas as pd
import pycountry_convert
import pycountry
import time
import wbgapi as wb
import warnings


try:
    import optuna
    optuna_available = True
except ImportError:
    warnings.warn(message="Unable to import optuna. Bayesian optimization is␣
↪not available.")
    optuna_available = False

try:
    import optunahub
    optunahub_available = True
except ImportError:
    warnings.warn(message="Unable to import optunahub. Auto-sampler not␣
↪available.")
    optunahub_available = False

plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300

DATA_DIR = 'data/'
IMG_DIR = 'images/'
MODELS_DIR = 'saved_models'
```

2 Define WorldBank Pipeline

[3]:
```
class WbDataPipeline():
    '''
    A class to pull and clean World Bank data for use in the happiness dataset
    '''

    def __init__(self, indicators, year, impute=True, missing_countries=False)␣
↪-> None:
        '''
        Creates the World Bank data pipeline

        Parameters:
        indicators (list): list of World Bank indicator codes
        year (int): year to pull data from. We used 2022
        impute (bool): whether to impute missing data
        missing_countries (bool): whether to include countries not in the␣
↪happiness dataset
        '''

        self.indicators = indicators
        self.year = year

        # Data from the happiness dataset had special formatting that we had to␣
↪extract by hand
        self.happiness_data = pd.read_csv(DATA_DIR + 'happiness/happiness.csv').
↪drop('Country', axis=1)
```

```
        self.valid_countries = self.happiness_data['ISO_A3'].unique()
        self.missing_countries = missing_countries
        self.impute = impute

        # Set the world bank database to the World Development Indicators
        wb.db = 2
        self.data = self.pull_data()

    def pull_data(self) -> pd.DataFrame:
        '''
        Pulls the World Bank data and merges it with the happiness data
        '''

        # Pull the data
        print(f"Pulling {len(self.indicators)} indicators from World Bank data..
↪.")
        features = wb.data.DataFrame(self.indicators, time=self.year)
        print("Done")
        features = features.reset_index()
        features = features.infer_objects()
        features = features.rename(columns={features.columns[0]: 'ISO_A3'})

        # If we only want countries in the happiness dataset
        if not self.missing_countries:
            features = features[features['ISO_A3'].isin(self.valid_countries)]

        if self.impute:
            features = self.impute_numeric_data(features)

        # Merge with happiness data
        if not self.missing_countries:
            merged = pd.merge(features, self.happiness_data, on='ISO_A3')
            return merged

        return features

    def impute_numeric_data(self, data) -> pd.DataFrame :
        '''This code fills in missing numerical data with the mean of its 5␣
↪nearest neighbors
        as determined by its nonmissing numerical data. No categorical features␣
↪are
        touched.

        Parameters:
        data (pd.DataFrame): the data to impute
        '''
        imputed_data = KNNImputer().fit_transform(data.iloc[:, 1:])
        data.iloc[:, 1:] = imputed_data
        return data

    def get_data(self) -> pd.DataFrame:
        '''
        Getter for the data
        '''

        if self.data is None:
            self.data = self.pull_data()
        return self.data

    def check_missing(self, threshold) -> pd.Series:
        '''
        Generates a report of how many features are less than a certain␣
↪percentage complete

        Parameters:
        threshold (float): the threshold for what percentage of the data must␣
↪be complete
        '''

        # Sum up the number of missing values
        nans = self.data.isna().sum()

        # We are more interested in the complete percentage, but this code was␣
↪originally written for missing percentage
        threshold = 1 - threshold
        nthreshold = np.round(self.data.shape[0] * threshold)
        cols = nans[nans > nthreshold]

        # Print the report
        print(f"The following features are less than {100*(1-threshold)}%␣
↪complete:")
        for col in cols.index:
            pcomplete = 1 - (nans[col] / self.data.shape[0])
            print(f"    {col}: {pcomplete*100}% complete")
        return cols

    def check_percent_complete(self) -> pd.Series:
        '''
        Returns the perentage of features that are complete
        '''

        nans = self.data.isna().sum()
        print(f"Percent complete: {self.get_percent_complete()}")
        return 1 - (nans / self.data.shape[0])
```

```
def clean_features(write_csv=None) -> pd.DataFrame:
    '''
    Function to get all of the feature data and generate csv containing
    the feature id, its description, and how complete that feature is.
    This is usefull for selecting features that dono have too many missing␣
 ↪values

    Parameters:
    write_csv (str): the file to write the data to
    '''

    entries = []
    indicators = wb.series.list()

    # Get all ids and values for all of the indicators
    ids = [(str(indicator['id']), str(indicator['value'])) for indicator in␣
 ↪indicators]

    # This can take a while so a progress bar is nice
    pbar = tqdm(total=len(ids), position=0, leave=True)

    for id, value in ids:
        dp = WbDataPipeline([id], 2022, impute=False)
        complete_percent = dp.get_percent_complete()[id]
        entries.append([id, value, complete_percent])
        pbar.update()
    pbar.close()

    # Return the data as a dataframe
    entries = pd.DataFrame(entries, columns=['id', 'value', 'complete_percent'])

    # Write the feature data to a file
    if write_csv is not None:
        entries.to_csv(write_csv)

    return entries

def generate_wb_dataset(complete_percent=1.0, write_csv=None,␣
 ↪missing_countries=False) -> pd.DataFrame:
    '''
    Function to generate a dataset from the World Bank data
    '''

    # Read from the feature data file
    feature_data = pd.read_csv('feature_data.csv')
    feature_data = feature_data.infer_objects()
    features = feature_data[feature_data['complete_percent'] >=␣
  ↪complete_percent]['id']

    # Get only countries satisfying the missing countries condition
    dp = WbDataPipeline(features, 2022, missing_countries=missing_countries)

    if write_csv is not None:
        dp.get_data().to_csv(write_csv)

    return dp.get_data()

# We encountered different naming conventions for countries in some of our␣
  ↪experimental datasets.
# This function standardizes some of thse names so we can use their ISO A3 codes

def name_change(df) -> pd.DataFrame:
        '''
        Replace unusual country names and get ISO A3 codes

        Parameters:
        df (pd.DataFrame): the dataframe
        '''

        # Define a dictionary of all the name changes we encountered
        name_changes = {'Bolivia (Plurinational State of)' : 'Bolivia',
        'Democratic Republic of the Congo' : 'Congo, The Democratic Republic of␣
  ↪the',
        'Iran (Islamic Republic of)':'Iran',
        'Micronesia (Federated States of)' : 'Micronesia, Federated States of',
        'Republic of Korea' : 'Korea, Republic of',
        'Swaziland' : 'Eswatini',
        'The former Yugoslav republic of Macedonia' : 'North Macedonia',
        'Turkey' : 'Türkiye',
        'Venezuela (Bolivarian Republic of)' : 'Venezuela, Bolivarian Republic␣
  ↪of',
        'Taiwan Province of China' : 'Taiwan',
        'Kosovo' : 'Serbia',
        'North Cyprus' : 'Cyprus',
        'Russia' : 'Russian Federation',
        'Hong Kong S.A.R. of China' : 'Hong Kong',
        'Ivory Coast' : 'CI',
        'Palestinian Territories' : 'PS',
        'Eswatini, Kingdom of' : 'SZ',}

        # Use pycountry to get the ISO A3 code for each country
        def get_country_code(country_name):
            if country_name[-1] == '*':
                country_name = country_name[:-1]
            if country_name in name_changes:
                country_name = name_changes[country_name]
            try:
                return pycountry.countries.get(country_name).alpha_3
            except:
                try:
                    return pycountry.countries.lookup(country_name).alpha_3
                except:
                    # If this is the case we need to manually enter the unusual␣
↪country name
                    raise ValueError(f"No ISO code associated with country␣
↪{country_name}")

        df['ISO_A3'] = df['Country'].apply(get_country_code)
        return df
```

3 Define Plotter for Happiness v. GDP

[4]:
```
def plot_gdp_happiness(ax, logscale=False) -> None:
    '''
    Generate the plot of GDP vs Happiness score

    Parameters:
    ax (matplotlib.pyplot.axis): the axis to plot on
    logscale (bool): whether to plot the data on a log scale
    '''

    # Pull the data
    wb.db = 2
    dp = WbDataPipeline(["NY.GDP.PCAP.PP.CD"], 2022, impute=False)
    data = dp.get_data()
    data = data.rename(columns={"NY.GDP.PCAP.PP.CD": "GDP"})

    # Manually enter the GDP for countries that are missing. Sourced from the␣
↪World Bank but not in the dataset
    data.loc[data["ISO_A3"] == "TKM", "GDP"] = 8792.55
    data.loc[data["ISO_A3"] == "VEN", "GDP"] = 3421
    data.loc[data["ISO_A3"] == "YEM", "GDP"] = 698.95

    # Whether to plot and fit models on log scale
    if logscale:
        data["GDP"] = np.log(data["GDP"])

    # Setup data for regression
    xs = np.linspace(data['GDP'].min(), data['GDP'].max(), 1000)
    X = data['GDP'].values.reshape(-1, 1)
    Y = data['Happiness score'].values.reshape(-1, 1)

    # Fit the linear model
    model1 = LinearRegression().fit(X, Y)
    lin_preds = model1.predict(X)
    print("Linear model MSE:", mean_squared_error(Y, lin_preds))
    print("Linear model R^2:", r2_score(Y, lin_preds))

    # Define a log model
    def log_model(th, x=X):
        return th[0] * np.log(x) + th[1]

    # Fit the log model
    res = minimize(lambda th : np.mean((log_model(th) - Y)**2), [1, 1])
    log_params = res.x
    log_preds = log_model(log_params)
    print("Log model MSE:", mean_squared_error(Y, log_preds))
    print("Log model R^2:", r2_score(Y, log_preds))

    # Plot all of the countries colored by continent
    colors = {'EU': 'blue', 'AS': 'orange', 'AF': 'red', 'NA': 'green', 'SA':␣
↪'yellow', 'OC': 'purple'}
    for country in data['ISO_A3']:
        color = colors[pycountry_convert.
↪country_alpha2_to_continent_code(pycountry.countries.get(alpha_3=country).
↪alpha_2)]
        ax.scatter(data.loc[data['ISO_A3'] == country, 'GDP'], data.
↪loc[data['ISO_A3'] == country, 'Happiness score'], marker='o', color=color,␣
↪s=4)

    # Plot the models
    ax.plot(xs, model1.predict(xs.reshape(-1,1)), color='black', label='Linear␣
↪Model')
    ax.plot(xs, log_model(log_params, xs.reshape(-1,1)), color='grey',␣
↪label='Log Model', linestyle='--')

    # Set the labels
    ax.set_xlabel('GDP per Capita (USD)', fontsize=6)
    ax.set_ylabel('Happiness Score', fontsize=6)
    ax.set_title('Happiness Score vs GDP per Capita', fontsize=10)
    ax.set_xticks

    # Create a custom legend
    custom_points = [Line2D([0], [0], color='black', label='Linear Model',␣
↪linestyle='-'),
                    Line2D([0], [0], color='grey', label='Log Model',␣
↪linestyle='--'),
                    Line2D([0], [0], color='red', marker='o', label='Africa',␣
↪linestyle=''),
                    Line2D([0], [0], color='orange', marker='o', label='Asia',␣
↪linestyle=''),
                    Line2D([0], [0], color='yellow', marker='o', label='Souh␣
↪America', linestyle=''),
                        Line2D([0], [0], color='green', marker='o',␣
↪label='North America', linestyle=''),
                        Line2D([0], [0], color='blue', marker='o',␣
↪label='Europe', linestyle=''),
                        Line2D([0], [0], color='purple', marker='o',␣
↪label='Oceania', linestyle=''),]
    ax.legend(handles=custom_points, loc="lower right", fontsize=6)
    ax.tick_params(axis='both', which='major', labelsize=6)
    ax.grid(True)
```

4 Define GeoPlotter for worldmap

[ ]:
```
class GeoPlotter():
    '''
    Class for plotting data on a world map
    '''

    def __init__(self, df=None) -> None:
        '''
        Initializes the GeoPlotter

        Parameters:
        df (pd.DataFrame): the dataframe to plot
        '''

        if 'ISO_A3' not in df.columns:
            raise ValueError("This dataframe does not have an ISO_A3 column.␣
↪Please use the DataPipeline change_name transform to prepare it for␣
↪plotting")
        else:
            self.df = df

        world = gpd.read_file(DATA_DIR + 'worldmap.gpkg')

        # Bug in geopandas see https://github.com/geopandas/geopandas/issues/
↪1041
        # These values are set wrong in the world map data file
        world.loc[world['NAME_EN'] == 'France', 'ISO_A3'] = 'FRA'
        world.loc[world['NAME_EN'] == 'Norway', 'ISO_A3'] = 'NOR'
        world.loc[world['NAME_EN'] == 'Somaliland', 'ISO_A3'] = 'SOM'
        world.loc[world['NAME_EN'] == 'Kosovo', 'ISO_A3'] = 'RKS'

        # Merge the data with the world map
        self.merged = pd.merge(world, self.df, on='ISO_A3')
        self.no_data = world[~world['ISO_A3'].isin(self.df['ISO_A3'])]

    def plot(self, col_name, ax=None, year=None, title=None, save_img:
↪bool=False, vmin:float=None, vmax:float=None, img_name:str='fig.pdf') ->␣
↪None:
        '''
        Plots the data from the specified column

        Parameters:
        col_name (str): the column to plot
        ax (matplotlib.pyplot.axis): the axis to plot on
        year (str): the year to plot from (we used 2022 in our experimets)
        title (str): the title of the plot
        '''

        if ax is None:
            fig, ax = plt.subplots(1, figsize=(10, 4))
        # Make sure data is set
        if year is None:
            df = self.merged
        else:
            df = self.merged[self.merged['Year'].astype(str) == year]

        if title is None:
            title = col_name

        # Plot the data and color missing countries grey
        df.plot(column=col_name, edgecolor="black", linewidth=0.2, ax=ax,␣
↪legend=True, vmin=vmin, vmax=vmax, legend_kwds={'label': 'Happiness Score',␣
↪'aspect': 30, 'shrink': 0.8})
        self.no_data.plot(ax=ax, edgecolor="grey", linewidth=0.2,␣
↪color='lightgrey', legend=True)

        # Format the plot
        ax.set_title(title, fontsize=16)
        ax.set_axis_off()
        fig = ax.get_figure()
        cax = fig.axes[1]
        cax.set_ylabel(col_name)

        if save_img:
            plt.savefig(img_name, format='pdf')

        plt.show()

    def animate(self, col_name, video_name)-> None:
        '''
        Animates updates of time series data. We never used this for the final␣
↪product, but we used it in experiments

        Parameters:
        col_name (str): the column to animate
        video_name (str): the name of the video to save
        '''
        fig, ax = plt.subplots(1, figsize=(10, 7))

        # Updata the animation for each year
        def update(year):
            df = self.merged[self.merged['Year'].astype(str) == str(year)]
            df.plot(column=col_name, ax=ax)
            ax.set_title(col_name + "\n" + str(year))

        # Write the video to file
        animation.writer = animation.writers['ffmpeg']
        ani = animation.FuncAnimation(fig, update,
            frames=sorted(set(self.df['Year'])),
            interval=self.df['Year'].max() - self.df['Year'].min() / 10)
        ani.save(video_name)
```

5 Define FeatureSeparator to get factors on happiness

[33]:
```
class FeatureSeparator():
    '''
    Class for determining what features influence happiness independently of GDP
    '''

    def __init__(self, data, similarity_metric="cos", alpha=1) -> None:
        '''
        Initializes the FeatureSeparator

        Parameters:
        data (pd.DataFrame): the data to analyze
        similarity_metric (str): the similarity metric to use. Either "cos" for␣
↪cosine similarity or "mi" for mutual information
        alpha (float): how much to weight the diffeence of GDP and happiness
        '''
        self.data = data
        # Need cosine similarity or mutual information
        if similarity_metric not in ["cos", "mi"]:
            raise ValueError("Invalid similarity metric. Use 'cos' for cosine␣
↪similarity or 'mi' for mutual information.")


        self.similarity_metric = similarity_metric

        if similarity_metric == "cos":
            self.similarity = np.matmul
        else:
            self.similarity = lambda X, Y : mutual_info_regression(X, Y,␣
↪random_state=3)

        self.alpha = alpha

    def get_separator(self, X, Y) -> np.array:
        '''
        Find a distribution D* that maximizes similarity with X and minimizes␣
↪similarity with Y

        Parameters:
        X (np.array): the first distribution
        Y (np.array): the second distribution
        '''

        # Normalize the data
        X = X.reshape(-1,1) / np.linalg.norm(X)
        Y = Y.reshape(-1,1) / np.linalg.norm(Y)

        if self.similarity_metric == "cos":
            X = np.ravel(X)
            Y = np.ravel(Y)

        # Define the objective function
        def obj(D):
            return -np.abs(self.similarity(X, D)) + self.alpha * np.abs(self.
↪similarity(Y, D))

        # Initial guess
        D = X.reshape(-1)
        D = D / np.linalg.norm(D)

        # Constrain the distribution to have norm 1 (mutual information depends␣
↪on parameter size)
        constraints = [{'type': 'eq', 'fun': lambda x: np.linalg.norm(x) - 1}]
        res = minimize(obj, D, constraints=constraints, tol=1e-14)

        # Print the similarity of the distributions
        print("Similarity of D with X", self.similarity(X, np.ravel(res.x)))
        print("Similarity of X with X", self.similarity(X, np.ravel(X)))
        return res.x

    def get_most_similar_feature(self, D, F, X, Y, n=None):
        '''
        Find which features are most similar to the separator D* in the dataset

        Parameters:
        D (np.array): the separator
        F (pd.DataFrame): the data to analyze
        X (np.array): the first distribution
        Y (np.array): the second distribution
        n (int): the number of features to return
        '''

        # Normalize the data
        if n is None:
            n = F.shape[1]
        F_arr = np.array(F)
        F_arr = F_arr / np.linalg.norm(F_arr, axis=0)

        if self.similarity_metric == "cos":
            F_arr = F_arr.T

        # Get the similarity of each feature to D*
        simD = self.similarity(F_arr, D)
        abs_simD = np.abs(simD)
        keys = np.argsort(abs_simD)[::-1][:n]

        simDs = simD[keys]
        simXs = self.similarity(F_arr, X)[keys]
        simYs = self.similarity(F_arr, Y)[keys]
        return F.columns[keys], simDs, simXs, simYs

    def get_separating_feature(self, col1, col2, n=None):
        '''
        Run the whole proceccs of finding D* and then the most similar features␣
↪to D*

        Parameters:
        col1 (str): the first column to compare
        col2 (str): the second column to compare
        n (int): the number of features to return
        '''
        data = self.data

        # Get the separator
        D = self.get_separator(data[col1].values, data[col2].values)
        F = data.drop([col1, col2], axis=1)
        X = data[col1]
        Y = data[col2]

        # Make sure the data is normalized
        X /= np.linalg.norm(X)
        Y /= np.linalg.norm(Y)

        # Get the most similar features
        return self.get_most_similar_feature(D, F, X, Y, n)
```

6 Define engineering to perform feature engineering & extraction

```
[ ]: class engineering():
         """This class is meant to perform feature engineering by either
         selecting the best features according to mutual information or
         generate new features using PCA (or a combination of both).

         Attributes:
             - X_tr: an array holding the training array of features
             - y_tr: an array holding the training targets
             - select_kfeats: a number specifying the number of features to
                              select
             - mut_info_kneighbors: the number of neighbors to use to compute
                                    mutual information
             - pca_comp: the number of principal components to compute
             - pca_desired_var: the desired variance the principal components
                                should have
             - X_tr_scaled_selected: an array holding the selected and scaled
                                     training array
             - features_selected: a list containing the names of the features
                                  selected
             - X_tr_pca: an array holding the transformed X_tr array into the
                         principal components containing only a certain number
                         of components needed to reach a desired variance.

         Methods:
             - __init__(): the constructor for the class
             - get_bestk_features(): method to select a certain number of features
             - get_pca_features(): method to select a certain number of
```

```
                              principal components
        - get_X_test_pca(): method to transform a test set of features into
                            the space of principal components


    Hidden Attributes:
        - _scaler: an instance of StandardScaler used to scale data. This
                   is fitted with data once select_kfeats or get_pca_feat-
                   res are called.
        - _pca: an instance of PCA used to compute principal components. It
                is fitted once get_pca_features is called.
        - _X_tr_scaled: an array holding the scaled X_tr array using _scaler
        - _selector: an instance of SelectKBest that is fitted at the time
                     select_features is called
        _ _pca_mask: a Boolean array holding the True values of the principal
                     components that are needed to achieve a desired variance

    Hidden Methods:
        - _mutual_scorer(): method to compute the mutual information
    """

    def __init__(self, X_tr:NDArray, y_tr:ArrayLike, select_kfeats:int=20,
                 mut_info_kneighbors:int=6, pca_comp:int=45,
                 pca_desired_var:float=0.95) -> None:

        """The constructor for the class. This functions accepts arguments
        and creates the attributes for the class.

        Parameters:
            - X_tr (NDArray): an array containing the features that will
                              be used for training.
            - y_tr (ArrayLike): an array containg the targets that will be
                                used for training.
            - select_kfeats (int): the number of features to select using
                                   mutual_info_regression. Defaulted to 20
            - mut_info_kneighbors (int): the number of neighbors (K nearest)
                                         to use for approximating mutual
                                         information. Defaulted to 6.
            - pca_comp (int): the number of principal components to compute
                              (using SVD). Defaulted to 45.
            - pca_desired_var (float): the desired variance that the chosen
                                       number principal components must have.
                                       Defauled to 0.95. Value must be bet-
                                       ween 0.1 and 1.

        Returns:
            None
        """

        self.X_tr = X_tr
        self.y_tr = y_tr
        self.select_kfeatures = select_kfeats
        self.mut_info_kneighbors = mut_info_kneighbors
        self.pca_components = pca_comp
        self.pca_desired_var = pca_desired_var
        self._scaler = StandardScaler()
        self._pca = PCA(n_components=pca_comp)

        # Check input
        if (pca_desired_var > 1) or (pca_desired_var < 0.1):
            raise ValueError("Desired PCA variance cannot be greater than 1 or␣
↪less than 0.1!")

    def _mutual_scorer(self, X:NDArray, y:ArrayLike) -> ArrayLike:
        """This function is meant to be called when selecting features
        according to mutual_info_regression. It allows the user to
        give the argument of number of neighbors to use.

        Parameters:
            - X (NDArray): an array X that will have its mutual informa-
                           tion to y computed
            - y (ArrayLike): an array y used as the target with which
                             to measure the mutual information of X

        Returns:
            - (ArrayLike): the computed mutual information estimation
        """

        return mutual_info_regression(X, y, n_neighbors=self.
↪mut_info_kneighbors)

    def get_bestk_features(self) -> None:
        """This function computes the mutual information between the stored
        training feature array X_tr and training target array y. It then
        selects a predetermined number of components and stores them as an
        attribute including the names.

        Parameters:
            - None

        Returns:
            - None
        """
        self._X_tr_scaled = self._scaler.fit_transform(X=self.X_tr)
        self._selector = SelectKBest(score_func=self._mutual_scorer, k=self.
↪select_kfeatures).fit(X=self._X_tr_scaled, y=self.y_tr)
        self.features_selected = self.X_tr.columns[self._selector.get_support()]
        self.X_tr_mut_info = self._X_tr_scaled[:, self._selector.get_support()]

    def get_X_test_mut_info(self, X_ts:NDArray) -> NDArray:
        """This function accepts an NDArray and returns the transformation of
        the array into the already created mutual info selector.

        Parameters:
            - X_ts (NDArray): an array containing the features that will be
                              used for testing.

        Returns:
            - (NDArray): the transformed test features arrays with features
                         already selected.
        """

        self._scaler.fit(X=self.X_tr)
        return self._selector.transform(X_ts)

    def get_pca_features(self) -> None:
        """This function performs PCA and saves the number of principal
        components needed to reach a predetermined desired variance. If
        it cannot reach the desired variance, the function raises an
        Exception.

        Parameters:
            - None

        Returns:
            - None
        """

        self._X_tr_scaled = self._scaler.fit_transform(X=self.X_tr)
        X_pca = self._pca.fit_transform(X=self._X_tr_scaled)
        cum_sum = np.cumsum(self._pca.explained_variance_ratio_)          # Get␣
↪the cumulative sum of the variance of each component

        # Check that the desired variance is actually met by the chosen number␣
↪of components
        if cum_sum.max() < self.pca_desired_var:
            raise Exception(f"Desired variance is {self.pca_desired_var} but␣
↪using {self.pca_components} components results in {cum_sum.max():.5f}")

        if cum_sum.min() > self.pca_desired_var:
            raise Exception(f"Minimum variance of PCA is {cum_sum.max():.5f}␣
↪which is greater than {self.pca_desired_var}. Please specify a greater value.
↪")

        self._pca_mask = (cum_sum >= self.pca_desired_var)                 #␣
↪Save the mask
        self.X_tr_pca = X_pca[:, ~self._pca_mask]          # Save the principal␣
↪components needed to achieve desired variance as an attribute

    def get_X_test_pca(self, X_ts:NDArray) -> NDArray:
        """This function accepts an NDArray and returns the transformation of
        the array into the already created principle component space.

        Parameters:
            - X_ts (NDArray): an array containing the features that will be
                              used for testing.

        Returns:
            - (NDArray): the transformed features array in the space of the
                         computed principal components
        """

        return (self._pca.transform(X=X_ts))[:, ~self._pca_mask]
```

7 Define `Model` Class for making ML Models

[ ]:
```
class Model():
    """This class is meant to create a user defined model. It allows the user␣
↪to specify
    the model type, estimator choice, hyperparameters, and hyperparameter␣
↪tuning strategy as
    well as other important options. The user can make various types of models␣
↪that can
    be used trained and hypertuned or just instantiated. See the docstring of␣
↪each method
    for more information.

    The class contains hidden attributes that are used to specify␣
↪hyperparameters or
    hyperparameter ranges or step sizes. These are divided by model choice and␣
↪have
    distinction by model choice. 'reg' is for regression and 'clf' is for␣
↪classification.
    There are also hidden methods that are used for hyperparameter tuning using␣
↪optuna.
```

```
    Wherever needed, estimator type is specified too.

    Refer to all of the docstrings for documentations on the class, attributes␣
↪methods,
    and hidden attributes for more information.

    Attributes
        - model_choice: the choice of model to make. The two options are
                            - 'clf' for classifier
                            - 'reg' for regression.
                        This is defaulted to 'reg'.
        - est_type: the type of estimator to use for a given model choice.
                    This is defaulted to 'rdf' for RandomForest. The
                    options are as follows:
                            - Classifiers (clf)
                                * 'rdf' for RandomForestClassifier (uses
                                oob_score)
                                * 'xgb' for  XGBClassifier
                            - Regression (reg):
                                * 'lin' for LinearRegression
                                * 'rdf' for RandomForestRegressor
                                * 'xgb' for XGBRegressor
                                * 'svr' for Support Vector Regressor
                                * 'ridge' for Ridge Regression
                                * 'lasso' for Lasso Regression
        - model_params: the parameters to give the model. Defaulted to None.
        - cv_fold: the number of folds to use in cross validation. Defa-
                 ulted to 2.
        - num_trials: the number of random samples or trials to use in
                    RandomizedSearchCV or optuna, respectively. Def-
                    aulted to 300.
        - n_jobs: the number of parallel jobs to run when hypertuning.
                Defaulted to -1 for all available cores.
        - tuning_strategy: the algorithm to use for hyperparameter tuning.
                           Defaulted to 'grid'. The options are as follows:
                            * 'grid' for GridSearchCV
                            * 'random' for RandomizedSearchCV
                            * 'bayesian' for Bayesian optimization using
                               TPESampler (Default of optuna)
                            * 'auto' for Auto-sampler (optunahub)
        - model: the model that is created and trained. If there is
               hypertuning, the best model is stored here.
        - best_params: the best hyperparameters found during hypertuning.
                     Defaulted to None if no hypertuning is done.
        - v_MSE: the best MSE (i.e. lowest) obtained on the validation set (for
               regression only)
        - v_r2: the best r-squared score obtained on the validation set (for
                 regression only)
        - t_MSE: the best MSE (i.e. lowest) obtained during the training phase␣
↪(for
                  regression only)
        - t_r2: the best r-squared score obtained during the training phase (for
                 regression only)

    Methods:
        - __init__(): the constructor for the class
        - make_full_model(): makes, trains, and hypertunes  a model on various
                             available hyperparameters. It uses the hidden
                             attributes to control model specifications and
                             doesn't require the user to pass in parameters to
                             access full hyperparameters.
        - save_model(): saves the made model to a joblib file.

    Hidden Attributes:

        - RandomForest:
            * _rdf_nestimators_range: the range of n_estimators to use.
                                      Defaulted to (100, 301)
            * _rdf_nestimators_step: the step size for n_estimators.
                                      Defaulted to 1
            * _rdf_maxdep_range: the range of max_depth to use. Defaulted
                                 to (4, 15)
            * _rdf_maxdep_step: the step size for max_depth. Defaulted
                                to 2
            * _rdf_minleaf_range: the range of min_samples_leaf to use.
                                  Defaulted to (2, 7)
            * _rdf_minleaf_step: the step size for min_samples_leaf.
                                 Defaulted to 1
            * _rdf_minsamples_range: the range of min_samples_split to use.
                                     Defaulted to (2, 7)
            * _rdf_minsamples_step: the step size for min_samples_split.
                                    Defaulted to 1
            * _rdf_maxfeat_range: the range of max_features to use.
                                  Defaulted to (4, 15)
            * _rdf_maxfeat_step: the step size for max_features. Defaulted
                                 to 1
            * _rdf_criterion_reg: the criterion to use for␣
↪RandomForestRegressor.
                                  Defaulted to "mse"  (mean squared error)

        - XGBoost:
            * _xgb_objective_clf: the objective for XGBoost classifier.
                                  Defaulted to "multi:softmax"
            * _xgb_num_classes_clf: the number of classes/labels for
```

```
                                XGBoost classifier. Defaulted to 9
          * _xgb_nestimators_range: the range of n_estimators to use.
                                  Defaulted to (100, 301)
          * _xgb_nestimators_step: the step size for n_estimators.
                                 Defaulted to 1
          * _gxb_eta_range: the range of eta/learning rate to use.
                          Defaulted to (0.001, 0.1)
          * _alpha_range: the range of alpha to use (L1 regularization).
                        Defaulted to (0.6, 10.1)
          * _lambda_range: the range of lambda to use (L2 regularization).
                         Defaulted to (0.6, 10.1)
          * _gamma_range: the range of gamma to use (minimum loss reduction
                        or penalty for many leaves). Defaulted to (0.6, 10.
↪1)
          * _xgb_max_depth_range: the range of max_depth to use. Defaulted
                                to (3, 10)
          * _xgb_max_depth_step: the step size for max_depth. Defaulted to 1
          * _xgb_objective_reg: the objective for XGBoost regressor. Defaulted
                              to "reg:squarederror"

      - Support Vector Regressor (SVR):
          * _svr_kernel: the kernel to use for SVR. Defaulted to "rbf".␣
↪Options
                        are as follows (from sklearn):
                            * 'linear' for linear kernel
                            * 'poly' for polynomial kernel
                            * 'rbf' for radial basis function kernel
                            * 'sigmoid' for sigmoid kernel
          * _svr_c_range: the range of C to use for SVR. Defaulted to (0.1,␣
↪10.5).
                        Note that strength of regularization is inversely␣
↪prop-
                        ortional to C.
          * _svr_epsilon_range: the range of epsilon to use for SVR.␣
↪Defaulted to
                              (0.01, 1.0)
          * _svr_gamma: the gamma parameter for the kernel. Defaulted to␣
↪"scale"
          * _svr_poly_degree_range: the range of polynomial degrees to use for
                                  SVR. Defaulted to (2, 5).
          * _svr_poly_degree_step: the step size for polynomial degrees.␣
↪Defaulted
                                  to 1
          * _svr_coef0_range: the range of coef0 to use for SVR. Defaulted to␣
↪(0.0, 5.0)
```

```
        - Ridge:
            * _ridge_alpha_range: a tuple for the range to find the best␣
↪regularization constant.
                                  Defaulted to (0.1, 10.5)
            * _ridge_max_iters: an int to specify the max number of iterations␣
↪to use for the
                                solver algorithm. Defaulted to 8000

        # Hidden attributes for Lasso Regression
        s._lasso_alpha_range = (0.1, 10.5)
        self._lasso_max_iters = 8000
        - Lasso:
            * _lasso_alpha_range: a tuple for the range to find the best␣
↪regularization constant.
                                  Defaulted to (0.1, 10.5)
            * _lasso_max_iters: an int to specify the max number of iterations␣
↪to use for convergence.
                                Defaulted to 8000

    Hidden Methods:
        - _rdf_obj(): a hidden method used to train and hypertune a RandomForest
                      model using optuna.
        - _get_rdf(): a hidden method used to get the best RandomForest model
                      after hypertuning.
        - _xgb_obj(): a hidden method used to train and hypertune a XGBoost
                      model using optuna.
        - _get_xgb(): a hidden method used to get the best XGBoost model after
                      hypertuning.
        - _svr_obj(): a hidden method used to train and hypertune a Support
                      Vector Regressor using optuna.
        - _get_svr(): a hidden method used to get the best Support Vector
                      Regressor model after hypertuning.
    """

    def __init__(self, model_choice: str = "reg", est_type: str = "rdf", params:
↪ Dict = None, cv_fold: int = 2,
                 tuning_strategy: str = "grid", num_trials: int = 300, n_jobs:␣
↪int = -1, **kwargs) -> None:
        """This function defines a user defined supervised learning model and␣
↪estimator
        according to the given input

        Parameters:
            - model_choice (str): the choice of model to make. Defaulted to␣
↪'reg'.
```

```
            - est_type (str): the type of estimator to use for a given model␣
↪choice.
                              Defaulted to 'rdf' for RandomForest.
            - params (Dict): the parameters to give the model. Defaulted to␣
↪None.
            - cv_fold (int): the number of folds to use when cross validating.
                             Defaulted to 2 for 2-fold cross validation.
            - tuning_strategy: the algorithm to use for hyperparameter tuning.
                               Defaulted to 'grid' for GridSearchCV.
            - num_trials (int): number of samples to use when performing baye-
                                sian optimization or randomized search.
            - n_jobs: the number of parallel jobs to run when hypertuning.
                      Defaulted to -1 for all available cores.
            - kwargs: keyword arguments meant for regression class

        Returns:
            - None
        """

        # Check user input
        if (model_choice.strip().lower() != "clf") and (model_choice.strip().
↪lower() != "reg"):
            raise ValueError(f"Model type must be either 'clf' for␣
↪classification or 'reg' for regression. Got {model_choice}")
        if (est_type.strip().lower() != 'rdf') and (est_type.strip().lower() !=␣
↪'xgb') and (
            est_type.strip().lower() != 'lin') and (est_type.strip().lower() !=␣
↪'svr') and (
            est_type.strip().lower() != 'ridge') and (est_type.strip().lower() !
↪= 'lasso'):
            raise ValueError(
                "Model type is not found. Please refer to the documentation to␣
↪choose and appropriate model.")
        if (cv_fold is None) or (cv_fold < 2):
            raise TypeError("cv_fold must be of type int that is greater than␣
↪or equal to 2.")
        if (tuning_strategy != "grid") and (tuning_strategy != "random") and (
            tuning_strategy != "bayesian") and (tuning_strategy != "auto"):
            raise TypeError("Hyperparameter tuning strategy must be either␣
↪'auto', 'bayesian', 'grid', or 'random'.")
        if not isinstance(num_trials, int):
            raise TypeError("num_trials must be of type int")
        if not isinstance(n_jobs, int):
            raise TypeError("n_jobs must be of type int")

        super(Model, self).__init__(**kwargs)
        # Define the attributes
        self.model_choice = model_choice.strip().lower()
        self.est_type = est_type.strip().lower()
        self.model_params = params
        self.cv_fold = cv_fold
        self.tuning_strategy = tuning_strategy.strip().lower()
        self.num_trials = num_trials
        self.n_jobs = n_jobs
        self.model = None

        # Hidden attributes for random forest (Hyperparameter tunining)
        self._rdf_nestimators_range = (25, 75)  # Uses np.arange (so go one␣
↪above)
        self._rdf_nestimators_step = 1
        self._rdf_maxdep_range = (5, 15)
        self._rdf_maxdep_step = 2
        self._rdf_minleaf_range = (3, 30)
        self._rdf_minleaf_step = 1
        self._rdf_minsamples_range = (2, 7)
        self._rdf_minsamples_step = 1
        self._rdf_maxfeat_range = (4, 15)
        self._rdf_maxfeat_step = 1
        self._rdf_criterion_reg = "squared_error"

        # Hidden attributes for xgboost
        self._xgb_objective_clf = "multi:softmax"
        self._xgb_num_classes_clf = 9
        self._xgb_nestimators_range = (25, 90)
        self._xgb_nestimators_step = 1
        self._gxb_eta_range = (0.001, 0.01)
        self._alpha_range = (0.6, 10.1)
        self._lambda_range = (0.6, 10.1)
        self._gamma_range = (0.6, 10.1)
        self._xgb_max_depth_range = (4, 15)
        self._xgb_max_depth_step = 1
        self._xgb_objective_reg = "reg:squarederror"

        # Hidden attributes for Support Vector Regressor (SVR)
        self._svr_kernel = "rbf"
        self._svr_c_range = (0.1, 10.5)
        self._svr_epsilon_range = (0.01, 1.0)
        self._svr_gamma_range = (0.1, 2.5)      # For rbf, poly, and sigmoid␣
↪kernels
        self._svr_poly_degree_range = (2, 5)    # Poly kernel parameters
        self._svr_poly_degree_step = 1
        self._svr_coef0_range = (0.1, 5.0)      # For poly, rbf, and sigmoid␣
↪kernels
```

```
        # Hidden attributes for Ridge Regression
        self._ridge_alpha_range = (0.1, 10.5)
        self._ridge_max_iters = 8000

        # Hidden attributes for Lasso Regression
        self._lasso_alpha_range = (0.1, 10.5)
        self._lasso_max_iters = 8000

    # Hidden methods for optuna hyperparameter tuning
    def _rdf_obj(self, trial:optuna.Trial, X_train:NDArray, y_train:ArrayLike)␣
↪-> float | Tuple[float|float]:
        """This function accepts a trial object and creates and trains a Random
        ForestClassifier with the specified hyperparameters as given by optuna
        using an optimization algorithm (be it TPESampler or Autosampler).

        Parameters:
            - trial (optuna.Trial): a specific trial object meant to signify the
                                    current trial/model optuna is training
            - X_train (NDArray): the training data
            - y_train (ArrayLike): the target data

        Returns:
            - (float): the oob_score for the classification algorithm
            OR
            - (float): the MSE on the oob-samples predictions (i.e. validation␣
↪set)
            - (float): the oob_score (r^2 as per sklearn docs) on the␣
↪oob-samples
                        prediction
        """

        if self.model_params is None:
            params = {"n_estimators": trial.suggest_int("n_estimators",␣
↪low=self._rdf_nestimators_range[0], high=self._rdf_nestimators_range[1],␣
↪step=self._rdf_nestimators_step),
                        "max_depth": trial.suggest_int("max_depth", low=self.
↪_rdf_maxdep_range[0], high=self._rdf_maxdep_range[1], step=self.
↪_rdf_maxdep_step),
                        "min_samples_leaf": trial.suggest_int("min_samples_leaf",␣
↪low=self._rdf_minleaf_range[0], high=self._rdf_minleaf_range[1], step=self.
↪_rdf_minleaf_step),
                        "min_samples_split": trial.suggest_int("min_samples_split",␣
↪low=self._rdf_minsamples_range[0], high=self._rdf_minsamples_range[1],␣
↪step=self._rdf_minsamples_step),
                    "max_features": trial.suggest_int("max_features", low=self.
↪_rdf_maxfeat_range[0],high=self._rdf_minsamples_range[1], step=self.
↪_rdf_maxfeat_step)
                    }
        else:
            params = self.model_params

        # Make the model
        if self.model_choice == "clf":
            params["criterion"] = trial.suggest_categorical("criterion",␣
↪["gini", "cross_entropy", "log_loss"]),
            model = RandomForestClassifier(**params, oob_score=True)
            scores = cross_val_score(model, X_train, y_train, scoring=lambda␣
↪est, X, y: est.oob_score_,
                                    n_jobs=self.n_jobs,
                                    cv=self.cv_fold)

            return scores.mean()

        # Regressor
        else:
            params["criterion"] = self._rdf_criterion_reg
            model = RandomForestRegressor(**params, oob_score=True, n_jobs=self.
↪n_jobs)
            #oob_scorer = lambda est, X, y: est.oob_score_
            #scoring = {"neg_mean_squared_error": "neg_mean_squared_error",
            #           "oob": oob_scorer}                                    ␣
↪                  # Make a metric out of the oob_score for CV
            #scores = cross_validate(estimator=model, X=X_train, y=y_train,␣
↪scoring=scoring, cv=self.cv_fold, n_jobs=self.n_jobs,␣
↪return_train_score=True)

            model.fit(X_train, y_train)
            mse = mean_squared_error(y_true=y_train, y_pred=model.
↪oob_prediction_)               # MSE on validation (oob)
            #scores = cross_val_score(model, X_train, y_train,␣
↪scoring='neg_mean_squared_error')
            #oobs = (scores['train_oob']).mean()
            #mses = -((scores['test_neg_mean_squared_error']).mean())

            return mse, model.oob_score_ #mses, oobs #-scores.mean()

    def _get_rdf(self, trial: optuna.Trial, X_train: NDArray,
                 y_train: ArrayLike) -> RandomForestClassifier |␣
↪RandomForestRegressor:
        """This is a helper function meant to accept the best optuna trial
        and return the best RandomForest model.

        Parameters:
            - trial (optuna.Trial): the best trial object from optuna
            - X_train (ArrayLike): the training data
            - y_train (ArrayLike): the target data

        Returns:
            - (RandomForestClassifier): the best RandomForestClassifier model
        """

        params = {"n_estimators": trial.suggest_int("n_estimators", low=self.
↪_rdf_nestimators_range[0], high=self._rdf_nestimators_range[1], step=self.
↪_rdf_nestimators_step),
                "max_depth": trial.suggest_int("max_depth", low=self.
↪_rdf_maxdep_range[0], high=self._rdf_maxdep_range[1], step=self.
↪_rdf_maxdep_step),
                "min_samples_leaf": trial.suggest_int("min_samples_leaf",␣
↪low=self._rdf_minleaf_range[0], high=self._rdf_minleaf_range[1], step=self.
↪_rdf_minleaf_step),
                "min_samples_split": trial.suggest_int("min_samples_split",␣
↪low=self._rdf_minsamples_range[0], high=self._rdf_minsamples_range[1],␣
↪step=self._rdf_minsamples_step),
                "max_features": trial.suggest_int("max_features", low=self.
↪_rdf_maxfeat_range[0],high=self._rdf_minsamples_range[1], step=self.
↪_rdf_maxfeat_step)
                }

        # Make the model
        if self.model_choice == "clf":
            params["criterion"] = trial.suggest_categorical("criterion",␣
↪["gini", "cross_entropy", "log_loss"]),
            model = RandomForestClassifier(**params, oob_score=True)
        else:
            params["criterion"] = self._rdf_criterion_reg
            model = RandomForestRegressor(**params, oob_score=True)

        model.fit(X_train, y_train)

        return model

    def _xgb_obj(self, trial: optuna.trial, X_train, y_train) ->␣
↪float|Tuple[float, float]:
        """This function accepts an optuna trial module that indicates the
        current trial of hyperparameter tuning and creates a XGBoost model to
        train. It returns the score after having trained the classifier or
        regressor. This function serves as a single call during each
        trial by the study object. The trial uses the TPESampler Bayesian
        or Autosampler optimization algorithm.

        Parameters:
            - trial (optuna.trial): the current trial of hyperparameter tuning
            - X_train (ArrayLike): the training data
            - y_train (ArrayLike): the target data

        Returns:
            - (float): the classifier score on the testing dataset
            OR
            - (float): the MSE on the validation set predictions
            - (float): the R^@ score on the validation set predictions
        """

        if self.model_params is None:
            params = {"n_estimators": trial.suggest_int("n_estimators",␣
↪low=self._xgb_nestimators_range[0], high=self._xgb_nestimators_range[1],␣
↪step=self._xgb_nestimators_step),
                      "eta": trial.suggest_float("eta", low=self.
↪_gxb_eta_range[0], high=self._gxb_eta_range[1], log=True),
                      "alpha": trial.suggest_float("alpha", low=self.
↪_alpha_range[0], high=self._alpha_range[1]),
                      "lambda": trial.suggest_float("lambda", low=self.
↪_lambda_range[0], high=self._lambda_range[1]),
                      "gamma": trial.suggest_float("gamma", low=self.
↪_gamma_range[0], high=self._gamma_range[1]),
                      "max_depth": trial.suggest_int("max_depth", low=self.
↪_xgb_max_depth_range[0], high=self._xgb_max_depth_range[1], step=self.
↪_xgb_max_depth_step),
                      }
        else:
            params = self.model_params

        # Make the model
        if self.model_choice == "clf":
            params["objective"] = self._xgb_objective_clf
            params["num_classes"] = self._xgb_num_classes_clf
            model = XGBClassifier(**params)
            scores = cross_val_score(model, X_train, y_train, n_jobs=self.
↪n_jobs, cv=self.cv_fold)
            mean = scores.mean()

            return mean
        else:
            params["objective"] = self._xgb_objective_reg
            model = XGBRegressor(**params)
            scoring = ('r2', 'neg_mean_squared_error')
            scores = cross_validate(estimator=model, X=X_train, y=y_train,␣
↪n_jobs=self.n_jobs, cv=self.cv_fold, scoring=scoring)
            #scores = cross_val_score(model, X_train, y_train, n_jobs=self.
↪n_jobs, cv=self.cv_fold, scoring='neg_mean_squared_error')
            mse = -(scores['test_neg_mean_squared_error'].mean())
            r2 = scores['test_r2'].mean()

            return mse, r2

    def _get_xgb(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> XGBClassifier | XGBRegressor:
        """This function accepts the best trial from optuna and returns the
        best XGBoost model according to the given hyperparameters.

        Parameters:
        - trial (optuna.Trial): the best trial from optuna
        - X_train (ArrayLike): the training data
        - y_train (ArrayLike): the target data

        Returns:
        - (XGBClassifier or XGBRegressor): the best XGBoost model
        """


        params = {"n_estimators": trial.suggest_int("n_estimators", low=self.
↪_xgb_nestimators_range[0], high=self._xgb_nestimators_range[1], step=self.
↪_xgb_nestimators_step),
                    "eta": trial.suggest_float("eta", low=self.
↪_gxb_eta_range[0], high=self._gxb_eta_range[1], log=True),
                    "alpha": trial.suggest_float("alpha", low=self.
↪_alpha_range[0], high=self._alpha_range[1]),
                    "lambda": trial.suggest_float("lambda", low=self.
↪_lambda_range[0], high=self._lambda_range[1]),
                    "gamma": trial.suggest_float("gamma", low=self.
↪_gamma_range[0], high=self._gamma_range[1]),
                    "max_depth": trial.suggest_int("max_depth", low=self.
↪_xgb_max_depth_range[0], high=self._xgb_max_depth_range[1], step=self.
↪_xgb_max_depth_step),
                    }

        # Make the model
        if self.model_choice == "clf":
            params["objective"] = self._xgb_objective_clf
            params["num_classes"] = self._xgb_num_classes_clf
            model = XGBClassifier(**params)
        else:
            params["objective"] = self._xgb_objective_reg
            model = XGBRegressor(**params)

        model.fit(X_train, y_train)

        return model

    def _svr_obj(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> float:
        """This function accepts a trial object and creates and trains a␣
↪Support Vector
        Regressor with the specified hyperparameters as given by optuna using␣
↪TPESampler
        or Autosampler from optuna.
        """

        if self.model_params is None:
            params = {
                      "C": trial.suggest_float("C", low=self._svr_c_range[0],␣
↪high=self._svr_c_range[1]),
                      "epsilon": trial.suggest_float("epsilon", low=self.
↪_svr_epsilon_range[0], high=self._svr_epsilon_range[1])
                      }
            params["kernel"] = self._svr_kernel

            if self._svr_kernel == "poly":
                params["degree"] = trial.suggest_int("degree", low=self.
↪_svr_poly_degree_range[0], high=self._svr_poly_degree_range[1], step=self.
↪_svr_poly_degree_step)
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])
                params["coef0"] = trial.suggest_float("coef0", low=self.
↪_svr_coef0_range[0], high=self._svr_coef0_range[1])

            elif self._svr_kernel == "sigmoid":
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])
                params["coef0"] = trial.suggest_float("coef0", low=self.
↪_svr_coef0_range[0], high=self._svr_coef0_range[1])

            elif self._svr_kernel == "rbf":
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])

        else:
            params = self.model_params

        # Make the model
        model = SVR(**params)

        scores = cross_val_score(model, X_train, y_train,␣
↪scoring="neg_mean_squared_error", n_jobs=self.n_jobs,
                                cv=self.cv_fold)

        return -scores.mean()

    def _get_svr(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> SVR:
        """This is a helper function meant to accept the best optuna trial
        and return the best Support Vector Regressor model.
        """

        params = {
                    "C": trial.suggest_float("C", low=self._svr_c_range[0],␣
↪high=self._svr_c_range[1]),
                    "epsilon": trial.suggest_float("epsilon", low=self.
↪_svr_epsilon_range[0], high=self._svr_epsilon_range[1])
                        }
        params["kernel"] = self._svr_kernel

        if self._svr_kernel == "poly":
                params["degree"] = trial.suggest_int("degree", low=self.
↪_svr_poly_degree_range[0], high=self._svr_poly_degree_range[1], step=self.
↪_svr_poly_degree_step)
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])
                params["coef0"] = trial.suggest_float("coef0", low=self.
↪_svr_coef0_range[0], high=self._svr_coef0_range[1])

        elif self._svr_kernel == "sigmoid":
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])
                params["coef0"] = trial.suggest_float("coef0", low=self.
↪_svr_coef0_range[0], high=self._svr_coef0_range[1])

        elif self._svr_kernel == "rbf":
                params["gamma"] = trial.suggest_float("gamma", low=self.
↪_svr_gamma_range[0], high=self._svr_gamma_range[1])

        # Make the model
        model = SVR(**params)

        model.fit(X_train, y_train)

        return model

    def _ridge_obj(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> float:
        """This function accepts a trial object and creates and trains a Ridge
        Regressor with the specified hyperparameters as given by optuna using
        an optuna optimization algorithm (Autosampler or TPESampler).
        """

        if self.model_params is None:
            params = {"alpha": trial.suggest_float("alpha", low=self.
↪_ridge_alpha_range[0], high=self._ridge_alpha_range[1]),
                      "solver": trial.suggest_categorical("solver", ["svd",␣
↪"cholesky", "lsqr", "sparse_cg", "sag", "saga"])
                      }
        else:
            params = self.model_params

        # Make the model
        model = Ridge(**params, max_iter=self._ridge_max_iters)

        scores = cross_val_score(model, X_train, y_train,␣
↪scoring="neg_mean_squared_error", n_jobs=self.n_jobs,
                                 cv=self.cv_fold)

        return -scores.mean()

    def _get_ridge(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> Ridge:
        """This is a helper function meant to accept the best optuna trial
        and return the best Ridge model.
        """
        params = {"alpha": trial.suggest_float("alpha", low=self.
↪_ridge_alpha_range[0], high=self._ridge_alpha_range[1]),
                  "solver": trial.suggest_categorical("solver", ["svd",␣
↪"cholesky", "lsqr", "sparse_cg", "sag", "saga"])
                  }
        # Make the model
        model = Ridge(**params, max_iter=self._ridge_max_iters)

        model.fit(X_train, y_train)

        return model

    def _lasso_obj(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> float:
        """This function accepts a trial object and creates and trains a Lasso
        Regressor with the specified hyperparameters as given by optuna using
        an optuna optimization algorithm (Autosampler or TPESampler).
        """

        if self.model_params is None:
            params = {"alpha": trial.suggest_float("alpha", low=self.
↪_lasso_alpha_range[0], high=self._lasso_alpha_range[1]),
                      }
        else:
            params = self.model_params

        # Make the model
        model = Lasso(**params, max_iter=self._lasso_max_iters)

        scores = cross_val_score(model, X_train, y_train,␣
↪scoring="neg_mean_squared_error", n_jobs=self.n_jobs,
                                 cv=self.cv_fold)

        return -scores.mean()

    def _get_lasso(self, trial: optuna.Trial, X_train: NDArray, y_train:␣
↪ArrayLike) -> Lasso:
        """This is a helper function meant to accept the best optuna trial
        and return the best Lasso model.
        """

        params = {"alpha": trial.suggest_float("alpha", low=self.
↪_lasso_alpha_range[0], high=self._lasso_alpha_range[1]),
                  }

        # Make the model
        model = Lasso(**params, max_iter=self._lasso_max_iters)

        model.fit(X_train, y_train)

        return model
    def make_full_model(self, X_train: NDArray, y_train: ArrayLike) -> None:
        """This methods makes a model that trains and hypertunes on
        all available hyperparameters. It uses the hidden attributes
        to control model specifications and doesn't require the user
        to pass in parameters to access full hyperparameters. It
        stores the model as an attribute.

        Parameters:
            - X_train (ArrayLike): the training data
            - y_train (ArrayLike): the target data
        """

        if self.model_params is not None:
            raise ValueError("Cannot make a model that trains and hypertunes on␣
↪all available hyperparameters " + \
                             "when given user defined parameters. Use the␣
↪hidden attributes to control model " + \
                             "specifications and don't pass in parameters to␣
↪access full hyperparameters.")

        warnings.warn(message=f"NOTE: Please check the hidden attributes for␣
↪the model choice: {self.model_choice}, and estimator: {self.est_type} before␣
↪hypertuning " +\
                              "should you want to have different␣
↪hyperparameters than the ones set as default.")
        time.sleep(3)
        print("Now continuing.")

        # Make grid search
        if self.tuning_strategy == "grid":

            # Make and hypertune classification models
            if self.model_choice == "clf":

                if self.est_type == "rdf":

                    print("Now making RandomForestClassifier...")
                    clf = RandomForestClassifier(oob_score=True)
                    parameters = {"n_estimators": [int(x) for x in np.
↪arange(*self._rdf_nestimators_range,
                                                                                   ␣
↪step=self._rdf_nestimators_step)],
                                  "criterion": ["gini", "cross_entropy",␣
↪"log_loss"],
                                  "max_depth": [int(x) for x in
```

```
                                                     np.arange(*self.
↪_rdf_maxdep_range, step=self._rdf_maxdep_step)],
                                  "min_samples_leaf": [int(x) for x in np.
↪arange(*self._rdf_minleaf_range,
                                                                              ␣
↪ step=self._rdf_minleaf_step)],
                                  "min_samples_split": [int(x) for x in np.
↪arange(*self._rdf_minsamples_range,
                                                                              ␣
↪  step=self._rdf_minsamples_step)]
                                  }
                  rdf_grid = GridSearchCV(estimator=clf,␣
↪param_grid=parameters, n_jobs=self.n_jobs, cv=self.cv_fold,
                                          scoring=lambda est, X, y: est.
↪oob_score_)

                  print("Training and hypertuning using Exhaustive Search...")
                  rdf_grid.fit(X_train, y_train)
                  print("Training completed.")

                  # Display and save best results
                  print(f"The best oob_score is {rdf_grid.best_score_}")
                  print("Best model and hyperparameters added as attribute to␣
↪class.")
                  self.model = rdf_grid.best_estimator_
                  self.best_params = rdf_grid.best_params_

              else:
                  print("Now making XGBClassifier...")
                  params = {"n_estimators": [int(x) for x in
                                                    np.arange(*self.
↪_xgb_nestimators_range, self._xgb_nestimators_step)],
                              "eta": [float(x) for x in np.linspace(*self.
↪_gxb_eta_range, num=100)],
                              "alpha": [float(x) for x in np.linspace(*self.
↪_alpha_range, num=100)],
                              "lambda": [float(x) for x in np.linspace(*self.
↪_lambda_range, num=100)],
                              "gamma": [float(x) for x in np.linspace(*self.
↪_gamma_range, num=100)],
                              "max_depth": [int(x) for x in
                                          np.arange(*self.
↪_xgb_max_depth_range, self._xgb_max_depth_step)],
                              "objective": self._xgb_objective_clf,
                              "num_classes": self._xgb_num_classes_clf
                              }
```

```
                    xgb_grid = GridSearchCV(estimator=XGBClassifier(),␣
↪param_grid=params, n_jobs=self.n_jobs,
                                            cv=self.cv_fold)

                    print("Training and hypertuning using Exhaustive Search...")
                    xgb_grid.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best oob_score is {xgb_grid.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = xgb_grid.best_estimator_
                    self.best_params = xgb_grid.best_params_

            # Make and hypertune regression models
            else:
                if self.est_type == "rdf":
                    print("Now making RandomForestRegressor...")
                    reg = RandomForestRegressor(oob_score=True)
                    parameters = {"n_estimators": [int(x) for x in np.
↪arange(*self._rdf_nestimators_range,
                                                                                  ␣
↪step=self._rdf_nestimators_step)],
                                  "criterion": ["squared_error"],
                                  "max_depth": [int(x) for x in
                                                np.arange(*self.
↪_rdf_maxdep_range, step=self._rdf_maxdep_step)],
                                  "min_samples_leaf": [int(x) for x in np.
↪arange(*self._rdf_minleaf_range,
                                                                                     ␣
↪ step=self._rdf_minleaf_step)],
                                  "min_samples_split": [int(x) for x in np.
↪arange(*self._rdf_minsamples_range,
                                                                                     ␣
↪  step=self._rdf_minsamples_step)],
                                  "max_features": [int(x) for x in
                                                   np.arange(*self.
↪_rdf_maxfeat_range, step=self._rdf_maxfeat_step)],
                                  }
                    rdf_grid = GridSearchCV(estimator=reg,␣
↪param_grid=parameters, n_jobs=self.n_jobs, cv=self.cv_fold,
                                            scoring='neg_mean_squared_error')

                    print("Training and hypertuning using Exhaustive Search...")
                    rdf_grid.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best oob_score is {rdf_grid.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = rdf_grid.best_estimator_
                    self.best_params = rdf_grid.best_params_

                elif self.est_type == "xgb":
                    print("Now making XGBRegressor...")
                    params = {"n_estimators": [int(x) for x in
                                                np.arange(*self.
↪_xgb_nestimators_range, self._xgb_nestimators_step)],
                              "eta": [float(x) for x in np.linspace(*self.
↪_gxb_eta_range, num=100)],
                              "alpha": [float(x) for x in np.linspace(*self.
↪_alpha_range, num=100)],
                              "lambda": [float(x) for x in np.linspace(*self.
↪_lambda_range, num=100)],
                              "gamma": [float(x) for x in np.linspace(*self.
↪_gamma_range, num=100)],
                              "max_depth": [int(x) for x in
                                            np.arange(*self.
↪_xgb_max_depth_range, self._xgb_max_depth_step)],
                              "objective": self._xgb_objective_reg,
                              }
                    xgb_grid = GridSearchCV(estimator=XGBRegressor(),␣
↪param_grid=params, n_jobs=self.n_jobs,
                                            cv=self.cv_fold)

                    print("Training and hypertuning using Exhaustive Search...")
                    xgb_grid.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best oob_score is {xgb_grid.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = xgb_grid.best_estimator_
                    self.best_params = xgb_grid.best_params_

                else:
                    print("Making making LinearRegression")
                    lin = LinearRegression(n_jobs=-1)
                    print("Training...")
                    lin.fit(X_train, y_train)
                    print("Training completed.")
                    print("Trained model saved as an attribute to class")
                    self.model = lin
                    self.best_params = None

        # Train and tune using randomized search
        elif self.tuning_strategy == "random":

            # Make and hypertune classification models
            if self.model_choice == "clf":

                if self.est_type == "rdf":
                    print("Now making RandomForestClassifier...")
                    clf = RandomForestClassifier(oob_score=True)
                    distribs = {"n_estimators": [int(x) for x in np.
↪arange(*self._rdf_nestimators_range,
                                                                                    ␣
↪step=self._rdf_nestimators_step)],
                                "criterion": ["gini", "cross_entropy",␣
↪"log_loss"],
                                "max_depth": [int(x) for x in
                                              np.arange(*self.
↪_rdf_maxdep_range, step=self._rdf_maxdep_step)],
                                "min_samples_leaf": [int(x) for x in
                                                     np.arange(*self.
↪_rdf_minleaf_range, step=self._rdf_minleaf_step)],
                                "min_samples_split": [int(x) for x in np.
↪arange(*self._rdf_minsamples_range,
                                                                                         ␣
↪step=self._rdf_minsamples_step)],
                                "max_features": [int(x) for x in
                                                 np.arange(*self.
↪_rdf_maxfeat_range, step=self._rdf_maxfeat_step)],
                                }
                    rdf_rand = RandomizedSearchCV(estimator=clf,␣
↪param_distributions=distribs, n_jobs=self.n_jobs,
                                                  cv=self.cv_fold,
                                                  n_iter=self.num_trials,␣
↪scoring=lambda est, X, y: est.oob_score_)

                    print("Training and hypertuning using Randomized Search...")
                    rdf_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best oob_score is {rdf_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = rdf_rand.best_estimator_
                    self.best_params = rdf_rand.best_params_

                else:
                    print("Now making XGBClassifier...")
                    params = {"n_estimators": [int(x) for x in
                                                   np.arange(*self.
↪_xgb_nestimators_range, self._xgb_nestimators_step)],
                              "eta": [float(x) for x in np.linspace(*self.
↪_gxb_eta_range, num=100)],
                              "alpha": [float(x) for x in np.linspace(*self.
↪_alpha_range, num=100)],
                              "lambda": [float(x) for x in np.linspace(*self.
↪_lambda_range, num=100)],
                              "gamma": [float(x) for x in np.linspace(*self.
↪_gamma_range, num=100)],
                              "max_depth": [int(x) for x in
                                                np.arange(*self.
↪_xgb_max_depth_range, self._xgb_max_depth_step)],
                              "objective": self._xgb_objective_clf,
                              "num_classes": self._xgb_num_classes_clf
                              }
                    xgb_rand = RandomizedSearchCV(estimator=XGBClassifier(),␣
↪param_distributions=params,
                                                   n_jobs=self.n_jobs,
                                                   cv=self.cv_fold, n_iter=self.
↪num_trials)

                    print("Training and hypertuning using Randomized Search...")
                    xgb_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best oob_score is {xgb_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = xgb_rand.best_estimator_
                    self.best_params = xgb_rand.best_params_

            else:
                if self.est_type == "rdf":
                    print("Now making RandomForestRegressor...")
                    reg = RandomForestRegressor(oob_score=True)
                    parameters = {"n_estimators": [int(x) for x in np.
↪arange(*self._rdf_nestimators_range,
                                                                          ␣
↪step=self._rdf_nestimators_step)],
                                  "criterion": ["squared_error"],
                                  "max_depth": [int(x) for x in
                                                np.arange(*self.
↪_rdf_maxdep_range, step=self._rdf_maxdep_step)],
                                  "min_samples_leaf": [int(x) for x in np.
↪arange(*self._rdf_minleaf_range,
                                                                             ␣
↪ step=self._rdf_minleaf_step)],
                                  "min_samples_split": [int(x) for x in np.
↪arange(*self._rdf_minsamples_range,
                                                                             ␣
↪  step=self._rdf_minsamples_step)],
                                  "max_features": [int(x) for x in
                                                   np.arange(*self.
↪_rdf_maxfeat_range, step=self._rdf_maxfeat_step)],
                                  }
                    rdf_rand = RandomizedSearchCV(estimator=reg,␣
↪param_distributions=parameters, n_jobs=self.n_jobs,
                                                 cv=self.cv_fold, n_iter=self.
↪num_trials,
                                                          ␣
↪scoring='neg_mean_squared_error')

                    print("Training and hypertuning using Randomized Search...")
                    rdf_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best MSE is {rdf_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = rdf_rand.best_estimator_
                    self.best_params = rdf_rand.best_params_

                elif self.est_type == "xgb":
                    print("Now making XGBRegressor...")
                    params = {"n_estimators": [int(x) for x in
                                               np.arange(*self.
↪_xgb_nestimators_range, self._xgb_nestimators_step)],
                              "eta": [float(x) for x in np.linspace(*self.
↪_gxb_eta_range, num=100)],
                              "alpha": [float(x) for x in np.linspace(*self.
↪_alpha_range, num=100)],
                              "lambda": [float(x) for x in np.linspace(*self.
↪_lambda_range, num=100)],
                              "gamma": [float(x) for x in np.linspace(*self.
↪_gamma_range, num=100)],
                              "max_depth": [int(x) for x in
                                            np.arange(*self.
↪_xgb_max_depth_range, self._xgb_max_depth_step)],
                              "objective": [self._xgb_objective_reg],
                              }
                    reg = XGBRegressor()
                    xgb_rand = RandomizedSearchCV(estimator=reg,␣
↪param_distributions=params,
                                                 n_jobs=self.n_jobs,
                                                 cv=self.cv_fold, n_iter=self.
↪num_trials)

                    print("Training and hypertuning using Randomized Search...")
                    xgb_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best MSE is {xgb_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = xgb_rand.best_estimator_
                    self.best_params = xgb_rand.best_params_

                elif self.est_type == 'svr':
                    print("Now making SVR...")
                    params = {"kernel": ["rbf", "poly", "linear", "sigmoid"],
                              "degree": [int(x) for x in np.arange(*self.
↪_svr_poly_degree_range, self._svr_poly_degree_step)],
                              "gamma": [float(x) for x in np.linspace(*self.
↪_svr_gamma_range, num=100)],
                              "C": [float(x) for x in np.linspace(*self.
↪_svr_c_range, num=100)],
                              "epsilon": [float(x) for x in np.linspace(*self.
↪_svr_epsilon_range, num=100)]
                              }
                    reg = SVR()
                    svr_rand = RandomizedSearchCV(estimator=reg,␣
↪param_distributions=params,
                                                 n_jobs=self.n_jobs,␣
↪scoring='neg_mean_squared_error',
                                                   cv=self.cv_fold, n_iter=self.
↪num_trials)

                    print("Training and hypertuning using Randomized Search...")
                    svr_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best MSE is {svr_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = svr_rand.best_estimator_
                    self.best_params = svr_rand.best_params_

                elif self.est_type == 'ridge':
                    print("Now making Ridge...")
                    params = {"alpha": [float(x) for x in np.linspace(*self.
↪_ridge_alpha_range, num=100)],
                              "solver": ["svd", "cholesky", "lsqr",␣
↪"sparse_cg", "sag", "saga"]
                              }
                    reg = Ridge()
                    ridge_rand = RandomizedSearchCV(estimator=reg,␣
↪param_distributions=params,
                                                   n_jobs=self.n_jobs,␣
↪scoring='neg_mean_squared_error',
                                                   cv=self.cv_fold, n_iter=self.
↪num_trials)

                    print("Training and hypertuning using Randomized Search...")
                    ridge_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best MSE is {ridge_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = ridge_rand.best_estimator_
                    self.best_params = ridge_rand.best_params_

                elif self.est_type == 'lasso':
                    print("Now making Lasso...")
                    params = {"alpha": [float(x) for x in np.linspace(*self.
↪_lasso_alpha_range, num=100)]
                              }
                    reg = Lasso()
                    lasso_rand = RandomizedSearchCV(estimator=reg,␣
↪param_distributions=params,
                                                    n_jobs=self.n_jobs,␣
↪scoring='neg_mean_squared_error',
                                                    cv=self.cv_fold, n_iter=self.
↪num_trials)

                    print("Training and hypertuning using Randomized Search...")
                    lasso_rand.fit(X_train, y_train)
                    print("Training completed.")

                    # Display and save best results
                    print(f"The best MSE is {lasso_rand.best_score_}")
                    print("Best model and hyperparameters added as attribute to␣
↪class.")
                    self.model = lasso_rand.best_estimator_
                    self.best_params = lasso_rand.best_params_

                else:
                    print("Making making LinearRegression")
                    lin = LinearRegression(n_jobs=-1)
                    print("Training...")
                    lin.fit(X_train, y_train)
                    print("Training completed.")
                    print("Trained model saved as an attribute to class")
                    self.model = lin
                    self.best_params = None

        # Train and tune using Bayesian optimization (using optuna TPESampler)
        elif self.tuning_strategy == "bayesian":
            if not optuna_available:
                raise Exception(
                    "optuna module is not available. Please install in order to␣
↪perform bayesian optimization.")
            optuna.logging.set_verbosity(optuna.logging.WARNING)

            # Make classifiers
            if self.model_choice == "clf":
                if self.est_type == "rdf":

                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _rdf_obj method.
                        Optuna requires a function call without any args"""
                        return self._rdf_obj(trial, X_train, y_train)
                    print("Making RandomForestClassifier using Bayesian␣
↪optimization...")
                    study = optuna.create_study(direction="maximize",␣
↪study_name="randfor_clf_tuning")
                    print("Training and tuning using Bayesian optimization...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best MSE is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_rdf(study.best_trial, X_train,␣
↪y_train)

                # XGBoost
                else:
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _xgb_obj method.
                        Optuna requires a function call without any args"""
                        return self._xgb_obj(trial, X_train, y_train)

                    print("Making XGBClassifier using Bayesian optimization...")
                    study = optuna.create_study(direction="maximize",␣
↪study_name="xgb_clf_tuning")
                    print("Training and tuning using Bayesian optimization...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_xgb(study.best_trial, X_train,␣
↪y_train)

            # Make regressors
            else:
                if self.est_type == "rdf":

                    print("Making RandomForestRegressor using Bayesian␣
↪optimization...")
                    study = optuna.create_study(directions=["minimize",␣
↪"maximize"], study_name="randfor_reg_tuning")  # Minimize the MSE and␣
↪maximize the oob_score (r^2)
                    #study = optuna.create_study(direction="minimize",␣
↪study_name="randfor_reg_tuning")
```

```
                print("Training and tuning...")
                study.optimize(lambda trial: self._rdf_obj(trial, X_train,␣
↪y_train), n_trials=self.num_trials, n_jobs=self.n_jobs,␣
↪show_progress_bar=True)
                print("Training completed.")

                best_trials = study.best_trials
                print(f"The number of best trials: {len(best_trials)}")
                print(f"Here is a list of the best trials of(MSE, R^2):")
                for i, trial in enumerate(best_trials):
                    print(f"\t*Trial {trial.number} (list index {i}):
↪\n\t\tParams: {trial.params}\n\t\tValues: {trial.values}")

                best_numb = int(input("Select the best trial index (i.e.␣
↪the index of the list): "))
                best_trial  = best_trials[best_numb]
                self.best_params = best_trial.params
                self.model = self._get_rdf(best_trial, X_train, y_train)
                self.v_MSE = best_trial.values[0]
                self.v_r2 = best_trial.values[1]
                self.t_MSE = mean_squared_error(y_true=y_train, y_pred=self.
↪model.predict(X=X_train))
                self.t_r2 = r2_score(y_true=y_train, y_pred=self.model.
↪predict(X=X_train))
                # self.model = self._get_rdf(study.best_trial, X_train,␣
↪y_train)
                # self.train_r2_score = r2_score(y_true=y_train,␣
↪y_pred=self.model.predict(X=X_train))
                # self.rdf_oob_mse = mean_squared_error(y_true=y_train,␣
↪y_pred=self.model.oob_prediction_)
                # self.rdf_oob_r2 = r2_score(y_true=y_train, y_pred=self.
↪model.oob_prediction_)

            elif self.est_type == "xgb":

                print("Making XGBRegressor using Bayesian optimization...")
                study = optuna.create_study(directions=["minimize",␣
↪"maximize"], study_name="xgb_reg_tuning")
                print("Training and tuning...")
                study.optimize(lambda trial: self._xgb_obj(trial, X_train,␣
↪y_train), n_trials=self.num_trials, n_jobs=self.n_jobs,␣
↪show_progress_bar=True)
                print("Training completed.")

                best_trials = study.best_trials
                print(f"The number of best trials: {len(best_trials)}")
```

```
                    print(f"Here is a list of the best trials of(MSE, R^2):")
                    for i, trial in enumerate(best_trials):
                        print(f"\t*Trial {trial.number} (list index {i}):
↪\n\t\tParams: {trial.params}\n\t\tValues: {trial.values}")

                    best_numb = int(input("Select the best trial index (i.e.␣
↪the index of the list): "))
                    best_trial  = best_trials[best_numb]
                    self.best_params = best_trial.params
                    self.model = self._get_xgb(best_trial, X_train, y_train)
                    self.v_MSE = best_trial.values[0]
                    self.v_r2 = best_trial.values[1]
                    self.t_MSE = mean_squared_error(y_true=y_train, y_pred=self.
↪model.predict(X=X_train))
                    self.t_r2 = r2_score(y_true=y_train, y_pred=self.model.
↪predict(X=X_train))

                elif self.est_type == "svr":

                    print("Making SVR with kernel "+self._svr_kernel+" using␣
↪Bayesian optimization...")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="svr_tuning")
                    print("Training and tuning...")
                    study.optimize(lambda trial: self._svr_obj(trial, X_train,␣
↪y_train), n_trials=self.num_trials, n_jobs=self.n_jobs,␣
↪show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.train_MSE = study.best_value
                    self.model = self._get_svr(study.best_trial, X_train,␣
↪y_train)
                    self.train_r2_score = r2_score(y_true=y_train, y_pred=self.
↪model.predict(X=X_train))

                elif self.est_type == "ridge":

                    print("Making Ridge using Bayesian optimization...")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="ridge_tuning")
                    print("Training and tuning...")
                    study.optimize(lambda trial: self._ridge_obj(trial,␣
↪X_train, y_train), n_trials=self.num_trials, n_jobs=self.n_jobs,␣
↪show_progress_bar=True)
```

```
                print("Training completed.")

                print(f"The best score is {study.best_value}")
                self.best_params = study.best_params
                self.train_MSE = study.best_value
                self.model = self._get_ridge(study.best_trial, X_train,␣
↪y_train)
                self.train_r2_score = r2_score(y_true=y_train, y_pred=self.
↪model.predict(X=X_train))

            elif self.est_type == "lasso":

                print("Making Lasso using Bayesian optimization...")
                study = optuna.create_study(direction="minimize",␣
↪study_name="lasso_tuning")
                print("Training and tuning...")
                study.optimize(lambda trial: self._lasso_obj(trial,␣
↪X_train, y_train), n_trials=self.num_trials, n_jobs=self.n_jobs,␣
↪show_progress_bar=True)
                print("Training completed.")

                print(f"The best score is {study.best_value}")
                self.best_params = study.best_params
                self.train_MSE = study.best_value
                self.model = self._get_lasso(study.best_trial, X_train,␣
↪y_train)
                self.train_r2_score = r2_score(y_true=y_train, y_pred=self.
↪model.predict(X=X_train))

            else:
                print("Making making LinearRegression")
                lin = LinearRegression(n_jobs=self.n_jobs)
                print("Training...")
                lin.fit(X_train, y_train)
                print("Training completed.")
                print("Trained model saved as an attribute to class")
                self.model = lin
                self.train_MSE = mean_squared_error(y_true=y_train,␣
↪y_pred=lin.predict(X=X_train))
                self.best_params = None
                self.train_r2_score = r2_score(y_true=y_train, y_pred=lin.
↪predict(X=X_train))

        # Train and tune using Autoensampler from optuna
        else:
            if not optuna_available:
                raise Exception(
                    "optuna module is not available. Please install in order to␣
↪use Autoensampler.")
            if not optunahub_available:
                raise Exception(
                    "auto-sampler module is not available. Please install in␣
↪order to use Autoensampler.")

            optuna.logging.set_verbosity(optuna.logging.WARNING)
            # Make classifiers
            if self.model_choice == "clf":
                if self.est_type == "rdf":

                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _rdf_obj method.
                        Optuna requires a function call without any args"""
                        return self._rdf_obj(trial, X_train, y_train)

                    print("Making RandomForestClassifier using Autosampler␣
↪optimization...")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="maximize",␣
↪study_name="randfor_clf_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best MSE is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_rdf(study.best_trial, X_train,␣
↪y_train)

                # XGBoost
                else:
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _xgb_obj method.
                        Optuna requires a function call without any args"""
                        return self._xgb_obj(trial, X_train, y_train)

                    print("Making XGBClassifier using Autosampler Optimization..
↪.")
```

```
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="maximize",␣
↪study_name="xgb_clf_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_xgb(study.best_trial, X_train,␣
↪y_train)

            # Make regressors
            else:

                if self.est_type == "rdf":
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _rdf_obj method.
                        Optuna requires a function call without any args"""
                        return self._rdf_obj(trial, X_train, y_train)

                    print("Making RandomForestRegressor using Autosampler␣
↪optimization...")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="randfor_reg_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")
                    best_trials = study.best_trials
                    print(f"The number of best trials: {len(best_trials)}")
                    for trial in best_trials:
                        print(f"Trial {trial.number}:\n\tParams: {trial.
↪params}\n\tValues: {trial.values}")
                    #self.best_params = study.best_params
                    #self.model = self._get_rdf(study.best_trial, X_train,␣
↪y_train)

                elif self.est_type == "xgb":
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _xgb_obj method.
                        Optuna requires a function call without any args"""
                        return self._xgb_obj(trial, X_train, y_train)

                    print("Making XGBRegressor using Autosampler optimization...
↪")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="xgb_reg_tuning",  sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_xgb(study.best_trial, X_train,␣
↪y_train)

                elif self.est_type == "svr":
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _svr_obj method.
                        Optuna requires a function call without any args"""
                        return self._svr_obj(trial, X_train, y_train)

                    print("Making SVR using Autosampler optimization...")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="svr_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_svr(study.best_trial, X_train,␣
↪y_train)

                elif self.est_type == "ridge":
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _ridge_obj method.
                        Optuna requires a function call without any args"""
                        return self._ridge_obj(trial, X_train, y_train)

                    print("Making Ridge using Autosampler optimization...")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="ridge_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_ridge(study.best_trial, X_train,␣
↪y_train)

                elif self.est_type == "lasso":
                    def obj(trial: optuna.Trial) -> float:
                        """This is a wrapper function n meant to call the␣
↪hidden _lasso_obj method.
                        Optuna requires a function call without any args"""
                        return self._lasso_obj(trial, X_train, y_train)

                    print("Making Lasso using Autosampler optimization...")
                    module = optunahub.load_module(package="samplers/
↪auto_sampler")
                    study = optuna.create_study(direction="minimize",␣
↪study_name="lasso_tuning", sampler=module.AutoSampler())
                    print("Training and tuning...")
                    study.optimize(lambda trial: obj(trial=trial),␣
↪n_trials=self.num_trials, n_jobs=-1, show_progress_bar=True)
                    print("Training completed.")

                    print(f"The best score is {study.best_value}")
                    self.best_params = study.best_params
                    self.model = self._get_lasso(study.best_trial, X_train,␣
↪y_train)

                else:
                    print("Making making LinearRegression")
                    lin = LinearRegression(n_jobs=-1)
                    print("Training...")
                    lin.fit(X_train, y_train)
                    print("Training completed.")
                    print("Trained model saved as an attribute to class")
                    self.model = lin
                    self.best_params = None

    def save_model(self, path_to_model:str) -> None:
        """This method saves the trained model to disk using the joblib library.
↪ NOTE:
        the extension for the filename must be of the form filename.sav (i.e.␣
↪it must
        end with .sav)

        Parameters:
            - path_to_model (str): the path, containing the filename, to save␣
↪the model to

        Returns:
            - None
        """

        if self.model is None:
            raise ValueError("Model has not yet been trained. Please train the␣
↪model and re-execute this method")

        joblib.dump(self.model, path_to_model)
        print("Model saved to "+path_to_model)
```

8 Define regression to make regressors

[ ]:
```
class regression(Model, GeoPlotter):
    """This class is meant to create any regression model as specified by the␣
↪user. It inherits from
    the Model and GeoPlotter classes.

    Attributes:
        - est_type (str): the type of estimator to use for regression. See␣
↪model documentation for all
                            available estimators.
        - tuning_strategy (str): the method to use for hyperparameter tuning.␣
↪See model documentation for
                                    all available tuning strategies.
        - num_trials (int): number of trials for tuning.
        - cv_fold (int): the number of folds to use for cross-validations.
        - n_jobs (int): the number of parallel jobs to run.
        - params (Dict): a dictionary of parameters to pass into the regression␣
↪estimator.
        - df (pd.DataFrame): the original dataframe (containing features, world␣
↪names, and targets)
        - happiness_predictions (ArrayLike): the happiness score predictions␣
↪given by the regression
                                            estimator.
        - happiness_residuals (ArrayLike): the residuals between the happiness␣
↪score predictions, as
                                          given by the used estimator, and the␣
↪actual happiness scores


    Methods:
        - __init__(): the constructor
        - get_happiness_predictions(): method to get the happiness score␣
↪predictions
        - get_happiness_residuals(): method to get the residuals of happiness␣
↪scores
        - get_worldplot(): the method to plot given data on a world plot
        - plot_line(): the method to plot given x,y data on a 2D plot
        - preds2csv(): the method to export given data to a csv file (using␣
↪pandas)
    """

    def __init__(self, est_type:str='rdf', tuning_strategy:str='grid',␣
↪num_trials:int=300, cv_fold:int=2,
                 n_jobs:int=-1, params:Optional[Dict]=None, og_df:pd.
↪DataFrame=None) -> None:
        """The constructor for the class. It set ups all attributes available␣
↪for the class.

        Parameters:
            - est_type (str): the type of estimator to use for regression.␣
↪Defaulted to 'rdf'. See model
                               documentation for all available estimators.
            - tuning_strategy (str): the method to use for hyperparameter␣
↪tuning. Defaulted to 'grid'.
                                    See model documentation for all all available␣
↪tuning strategies.
            - num_trials (int): number of trials for tuning. Defaulted to 300
            - cv_fold (int): the number of folds to use for cross-validations.␣
↪Defaulted to 4.
            - n_jobs (int): the number of parallel jobs to run. Defaulted to -1␣
↪for all available processors.
```

```
            - params (Dict): a dictionary of parameters to pass into the␣
↪regression estimator. Defaulted to
                                 None.
            - og_df (pd.DataFrame): the original dataframe unaltered from where␣
↪all information was used to
                                        make the ML model (containing features,␣
↪world names, and targets)

        Returns:
            - None
        """

        super(regression, self).__init__(est_type=est_type,␣
↪tuning_strategy=tuning_strategy, num_trials=num_trials,
                                  cv_fold=cv_fold, n_jobs=n_jobs, params=params,␣
↪df=og_df)
        self.happiness_predictions = None
        self.happiness_residuals = None

    def get_happiness_predictions(self, X:NDArray, y_true:ArrayLike) -> None:
        """This function accepts a set data features and stores the predictions
        of happiness-scores made by the model as an attributes. It also stores
        the MSE and r-squared scores obtained from making the predictions.

        Parameters:
            - X (ArrayLike): the set of data features
            - y_true (ArrayLike): the true happiness targets. Used to get
                                  MSE and r-squared scores

        Returns:
            - None
        """

        if self.model is None:
            raise Exception("Model has not yet been trained. Please train the␣
↪model and re-execute this method")

        self.happiness_predictions = self.model.predict(X)
        self.happiness_predictions_MSE = mean_squared_error(y_true=y_true,␣
↪y_pred=self.happiness_predictions)
        self.happiness_predictions_r2_score = r2_score(y_true=y_true,␣
↪y_pred=self.happiness_predictions)

    def get_happiness_residuals(self, y_true:ArrayLike) -> None:
        """This function accepts the true target values and stores the
        residuals of the model.
        Parameters:
            - y_true (ArrayLike): the true target values

        Returns:
            - None
        """

        if self.model is None:
            raise Exception("Model has not yet been trained. Please train the␣
↪model and re-execute this method")
        elif self.happiness_predictions is None:
            raise Exception("Model has not yet computed happiness-score␣
↪predictions. Compute the predictions using " +\
                            "'.get_happiness_predictions(X)' then rerun this␣
↪method.")

        self.residuals = y_true - self.happiness_predictions

    def get_worldplot(self, y_data:ArrayLike=[], y_name:str="happiness␣
↪predictions", fig_title:str='title',
                      save_fig:bool=False, path_to_fig:str='fig.pdf') -> None:
        """This function plots the given data into the world plot. It then␣
↪saves the
        created image, if specified, into a pdf format.

        Parameters:
            - y_data (ArrayLike): the data to plot on the world map. Defaulted
                                  to an empty list. Default value will plot
                                  the happiness predictions.
            - y_name (str): the name of the column of the original dataframe
                            to plot on the worldmap. Defaulted to 'happiness
                            predictions'.
            - fig_title (str): the title to give the image. Defaulted to␣
↪'title'.
            - save_fig (bool): whether to save the figure into a pdf format
                               or not. Defaulted to False
            - path_to_fig (str): the path, containing the filename, on where to␣
↪save
                                 the figure. Defauled to 'fig.pdf'

        Returns:
            - None
        """

        if y_name.strip().lower() == "happiness predictions":
            y_data = self.happiness_predictions

        if (y_name.strip().lower() != "happiness predictions") and not y_data:
            raise ValueError("Argument 'y_data' cannot be empty if not using␣
↪default 'happiness prediction'.")

        self.df[y_name] = y_data
        obj = GeoPlotter(df=self.df)
        obj.plot(col_name=y_name, title=fig_title, save_img=save_fig,␣
↪img_name=path_to_fig)

    def plot_line(self, y_data:ArrayLike, x_data:ArrayLike, xlabel:str='x',␣
↪ylabel:str='y', title:str='Regression',
                  save_line_plot:bool=False, path_to_fig:str='fig.pdf') -> None:
        """This function plots the given arrays in 2D plot. It also saves the␣
↪image if
        specified in pdf format.

        Parameters:
            - y_data (ArrayLike): the data to plot on the y-axis
            - x_data (ArrayLike): the data to plot on the x-axis
            - xlabel (str): the label for the x-axis. Defaulted to 'x'
            - ylabel (str): the label for the y-axis. Defaulted to 'y'
            - title (str): the title for the created image. Defaulted to␣
↪'Regression'.
            - save_line_plot (bool): whether to save the created line plot or␣
↪not. De-
                                            faulted to False.
            - path_to_fig (str): the path, containing the filename, on where to␣
↪save
                                    the figure. Defaulted to 'fig.pdf'.

        Returns:
            - None
        """

        ax = plt.subplot(111)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)
        ax.plot(x_data, y_data)

        if save_line_plot:
            plt.savefig(path_to_fig, format='pdf')

        plt.show()
    def preds2csv(self, preds:ArrayLike|NDArray=[], header:bool|List|str=False,
                  path_to_file:str="./happiness/data/rdf_reg_happ_preds.csv") ->␣
↪None:
        """This function exports given data to a csv file. By default, it␣
↪exports the
        happiness scores predictions if the 'preds' argument is empty.

        Parameters:
            - preds (ArrayLike|NDArray): the data to export to csv. Defaulted␣
↪to an
                                        empty list.
            - header (bool|List|str): the header to give the csv file.␣
↪Defaulted to False.
                                      It can be a boolean, a list, or a string.
            - path_to_file (str): the path, including file name, to the␣
↪directory in
                                  where to store the exported csv file.␣
↪Defaulted to
                                  './data/happiness/rdf_reg_happ_preds.csv'

        Returns:
            - None
        """

        if len(preds) == 0:

            # Check for predictions
            if "happiness predictions" not in self.df.columns.to_list():
                self.df["happiness predictions"] = self.happiness_predictions

            df = self.df[['ISO_A3', "happiness predictions"]]
            df.to_csv(path_to_file, header=False)

        else:
            df = pd.DataFrame(preds)
            df.to_csv(path_to_file, header=header)
```

9 Loading a trained model

```
[ ]: class load_model():
         """This class is meant to load and help in using a trained ML model.

         Attributes:
             - model: the loaded trained model
```

```
        - df: original unaltered dataframe from where the training and testing␣
↪data come from
        - happiness_predictions: the happiness score predictions made by the␣
↪model
        - residuals: the residuals between the happiness predictions and the␣
↪true happiness
                        scores

    Methods:
        - __init__(): the constructor for the class
        - get_happiness_predictions(): the method used to get happiness␣
↪predictions
        - get_happiness_residuals(): the method used to get happiness residuals
        - get_worldplot(): the method used to create a world plot
        - preds2csv(): the method used to export data to a csv file
    """


    def __init__(self, path_to_model:str=MODELS_DIR+'./saved_models/rdf_reg_new.
↪sav', og_df:pd.DataFrame=[]):
        """This function initiates the class and uploads the saved model as an␣
↪attribute.

        Parameters:
            - path_to_model (str): the path to the joblib (.sav) file␣
↪containing the trained model. Defaul-
                                    ted to MODELS_DIR+'rdf_reg_new.sav.sav' for␣
↪a random forest regressor.
            - og_df (pd.DataFrame): the original dataframe unaltered from where␣
↪all information was used to
                                    make the ML model (containing features,␣
↪world names, and targets). Def-
                                    aulted to an empty list.

        Returns:
            - None
        """

        self.model = joblib.load(path_to_model)
        if isinstance(og_df, list):
            raise ValueError("Must give the original dataframe when␣
↪instatiating (i.e. constructor)!")
        self.df = og_df

    def get_happiness_predictions(self, X:NDArray|pd.DataFrame) -> None:
        """This function accepts a set data features and stores the predictions
        of happiness-scores made by the model as an attributes

        Parameters:
            - X (ArrayLike|pd.DataFrame): the set of data features

        Returns:
            - None
        """

        self.happiness_predictions = self.model.predict(X)

    def get_happiness_residuals(self, y_true:ArrayLike) -> None:
        """This function accepts the true target values and stores the
        residuals of the model.

        Parameters:
            - y_true (ArrayLike): the true target values

        Returns:
            - None
        """

        if self.happiness_predictions is None:
            raise Exception("Model has not yet computed happiness-score␣
↪predictions. Compute the predictions using " +\
                            "'.get_happiness_predictions(X)' then rerun this␣
↪method.")

        self.residuals = y_true - self.happiness_predictions

    def get_worldplot(self, y_data:ArrayLike=[], y_name:str="happiness␣
↪predictions", fig_title:str='title',
                      save_fig:bool=False, path_to_fig:str=IMG_DIR+'fig.pdf')␣
↪-> None:
        """This function plots the given data into the world plot. It then␣
↪saves the
        created image, if specified, into a pdf format.

        Parameters:
            - y_data (ArrayLike): the data to plot on the world map. Defaulted
                                  to an empty list. Default value will plot
                                  the happiness predictions.
            - y_name (str): the name of the column of the original dataframe
                            to plot on the worldmap. Defaulted to 'happiness
                            predictions'.
            - fig_title (str): the title to give the image. Defaulted to␣
↪'title'.
```

```
            - save_fig (bool): whether to save the figure into a pdf format
                                   or not. Defaulted to False
            - path_to_fig (str): the path, containing the filename, on where to␣
↪save
                                     the figure. Defauled to IMG_DIR+'fig.pdf'

        Returns:
            - None
        """

        if self.happiness_predictions is None:
            raise Exception("Model has not yet computed happiness-score␣
↪predictions. Compute the predictions using " +\
                             "'.get_happiness_predictions(X)' then rerun this␣
↪method.")

        if y_name.strip().lower() == "happiness predictions":
            y_data = self.happiness_predictions

        if (y_name.strip().lower() != "happiness predictions") and not y_data:
            raise ValueError("Argument 'y_data' cannot be empty if not using␣
↪default 'happiness prediction'.")

        self.df[y_name] = y_data
        obj = GeoPlotter(df=self.df)
        obj.plot(col_name=y_name, title=fig_title, save_img=save_fig,␣
↪img_name=path_to_fig)

    def preds2csv(self, preds:ArrayLike|NDArray=[], header:bool|List|str=False,
                  path_to_file:str=DATA_DIR+"rdf_reg_happ_preds.csv") -> None:
        """This function exports given data to a csv file. By default, it␣
↪exports the
        happiness scores predictions if the 'preds' argument is empty.

        Parameters:
            - preds (ArrayLike|NDArray): the data to export to csv. Defaulted␣
↪to an
                                        empty list.
            - header (bool|List|str): the header to give the csv file. Defaulte␣
↪to False.
                                     It can be a boolean, a list, or a string.
            - path_to_file (str): the path, including file name, to the␣
↪directory in
                                  where to store the exported csv file.␣
↪Defaulted to
                                  DATA_DIR+'rdf_reg_happ_preds.csv'
        Returns:
            - None
        """

        if len(preds) == 0:

            # Check for predictions
            if "happiness predictions" not in self.df.columns.to_list():
                self.df["happiness predictions"] = self.happiness_predictions

            df = self.df[['ISO_A3', "happiness predictions"]]
            df.to_csv(path_to_file, header=False)

        else:
            df = pd.DataFrame(preds)
            df.to_csv(path_to_file, header=header)
```