C2M2_Assignment

January 5, 2021

1 Risk Models Using Tree-based Models

Welcome to the second assignment of Course 2!

1.1 Outline

- Section **??**
- Section **??**
- Section **??**
- Section **??**
    - Section **??**
- Section **??**
    - Section **??**
- Section **??**
    - Section **??**
- Section **??**
- Section **??**
    - Section **??**
- Section **??**
    - Section **??**
    - Section **??**
- Section **??**
- 11. Explanations: SHAP

In this assignment, you'll gain experience with tree based models by predicting the 10-year risk of death of individuals from the NHANES I epidemiology dataset (for a detailed description of this dataset you can check the CDC Website). This is a challenging task and a great test bed for the machine learning methods we learned this week.

As you go through the assignment, you'll learn about:

- Dealing with Missing Data
    - Complete Case Analysis.
    - Imputation
- Decision Trees
    - Evaluation.
    - Regularization.
- Random Forests

– Hyperparameter Tuning.

1. Import Packages

We'll first import all the common packages that we need for this assignment.

- `shap` is a library that explains predictions made by machine learning models.
- `sklearn` is one of the most popular machine learning libraries.
- `itertools` allows us to conveniently manipulate iterable objects such as lists.
- `pydotplus` is used together with `IPython.display.Image` to visualize graph structures such as decision trees.
- `numpy` is a fundamental package for scientific computing in Python.
- `pandas` is what we'll use to manipulate our data.
- `seaborn` is a plotting library which has some convenient functions for visualizing missing data.
- `matplotlib` is a plotting library.

```
[1]: import shap
import sklearn
import itertools
import pydotplus
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from IPython.display import Image

from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer, SimpleImputer

# We'll also import some helper functions that will be useful later on.
from util import load_data, cindex
```

2. Load the Dataset

Run the next cell to load in the NHANES I epidemiology dataset. This dataset contains various features of hospital patients as well as their outcomes, i.e. whether or not they died within 10 years.

```
[2]: X_dev, X_test, y_dev, y_test = load_data(10)
```

The dataset has been split into a development set (or dev set), which we will use to develop our risk models, and a test set, which we will use to test our models.

We further split the dev set into a training and validation set, respectively to train and tune our models, using a 75/25 split (note that we set a random state to make this split repeatable).

```
[3]: X_train, X_val, y_train, y_val = train_test_split(X_dev, y_dev, test_size=0.25,␣
     ↪random_state=10)
```

3. Explore the Dataset

The first step is to familiarize yourself with the data. Run the next cell to get the size of your training set and look at a small sample.

```
[4]: print("X_train shape: {}".format(X_train.shape))
     X_train.head()
```

X_train shape: (5147, 18)

[4]:

	Age	Diastolic BP	Poverty index	Race	Red blood cells \
1599	43.0	84.0	637.0	1.0	49.3
2794	72.0	96.0	154.0	2.0	43.4
1182	54.0	78.0	205.0	1.0	43.8
6915	59.0	90.0	417.0	1.0	43.4
500	34.0	80.0	385.0	1.0	77.7

	Sedimentation rate	Serum Albumin	Serum Cholesterol	Serum Iron \
1599	10.0	5.0	253.0	134.0
2794	23.0	4.3	265.0	106.0
1182	12.0	4.2	206.0	180.0
6915	9.0	4.5	327.0	114.0
500	9.0	4.1	197.0	64.0

	Serum Magnesium	Serum Protein	Sex	Systolic BP	TIBC	TS \
1599	1.59	7.7	1.0	NaN	490.0	27.3
2794	1.66	6.8	2.0	208.0	301.0	35.2
1182	1.67	6.6	2.0	NaN	363.0	49.6
6915	1.65	7.6	2.0	NaN	347.0	32.9
500	1.74	7.3	2.0	NaN	376.0	17.0

	White blood cells	BMI	Pulse pressure
1599	9.1	25.803007	34.0
2794	6.0	33.394319	112.0
1182	5.9	20.278410	34.0
6915	6.1	32.917744	78.0
500	8.2	30.743489	30.0

Our targets y will be whether or not the target died within 10 years. Run the next cell to see the target data series.

```
[5]: y_train.head(20)
```

[5]: 1599   False
     2794   True

3

```
1182     False
6915     False
500      False
1188      True
9739     False
3266     False
6681     False
8822     False
5856      True
3415     False
9366     False
7975     False
1397     False
6809     False
9461     False
9374     False
1170      True
158      False
Name: time, dtype: bool
```

Use the next cell to examine individual cases and familiarize yourself with the features.

```
[6]: i = 10
     print(X_train.iloc[i,:])
     print("\nDied within 10 years? {}".format(y_train.loc[y_train.index[i]]))
```

```
Age                     67.000000
Diastolic BP            94.000000
Poverty index          114.000000
Race                     1.000000
Red blood cells         43.800000
Sedimentation rate      12.000000
Serum Albumin            3.700000
Serum Cholesterol      178.000000
Serum Iron              73.000000
Serum Magnesium          1.850000
Serum Protein            7.000000
Sex                      1.000000
Systolic BP            140.000000
TIBC                   311.000000
TS                      23.500000
White blood cells        4.300000
BMI                     17.481227
Pulse pressure          46.000000
Name: 5856, dtype: float64

Died within 10 years? True
```

4. Dealing with Missing Data

Looking at our data in `X_train`, we see that some of the data is missing: some values in the output of the previous cell are marked as `NaN` ("not a number").

Missing data is a common occurrence in data analysis, that can be due to a variety of reasons, such as measuring instrument malfunction, respondents not willing or not able to supply information, and errors in the data collection process.

Let's examine the missing data pattern. `seaborn` is an alternative to `matplotlib` that has some convenient plotting functions for data analysis. We can use its `heatmap` function to easily visualize the missing data pattern.

Run the cell below to plot the missing data:

```
[7]:  sns.heatmap(X_train.isnull(), cbar=False)
      plt.title("Training")
      plt.show()

      sns.heatmap(X_val.isnull(), cbar=False)
      plt.title("Validation")
      plt.show()
```

Validation
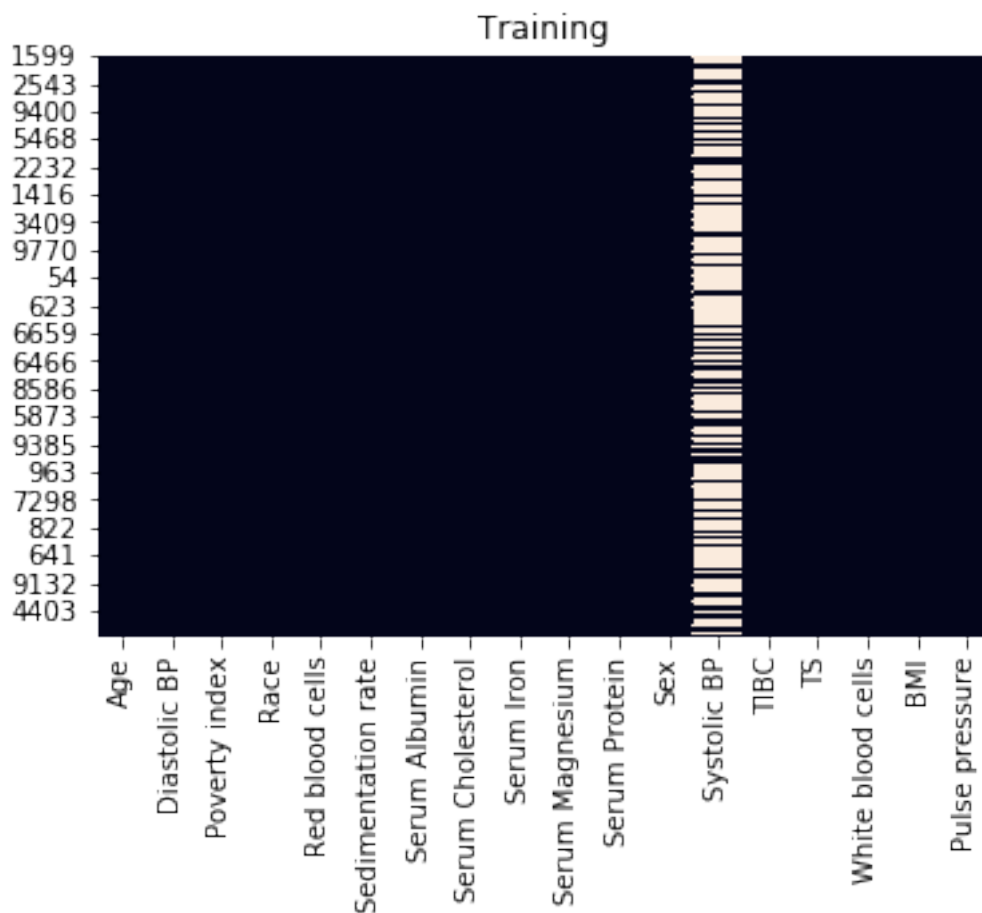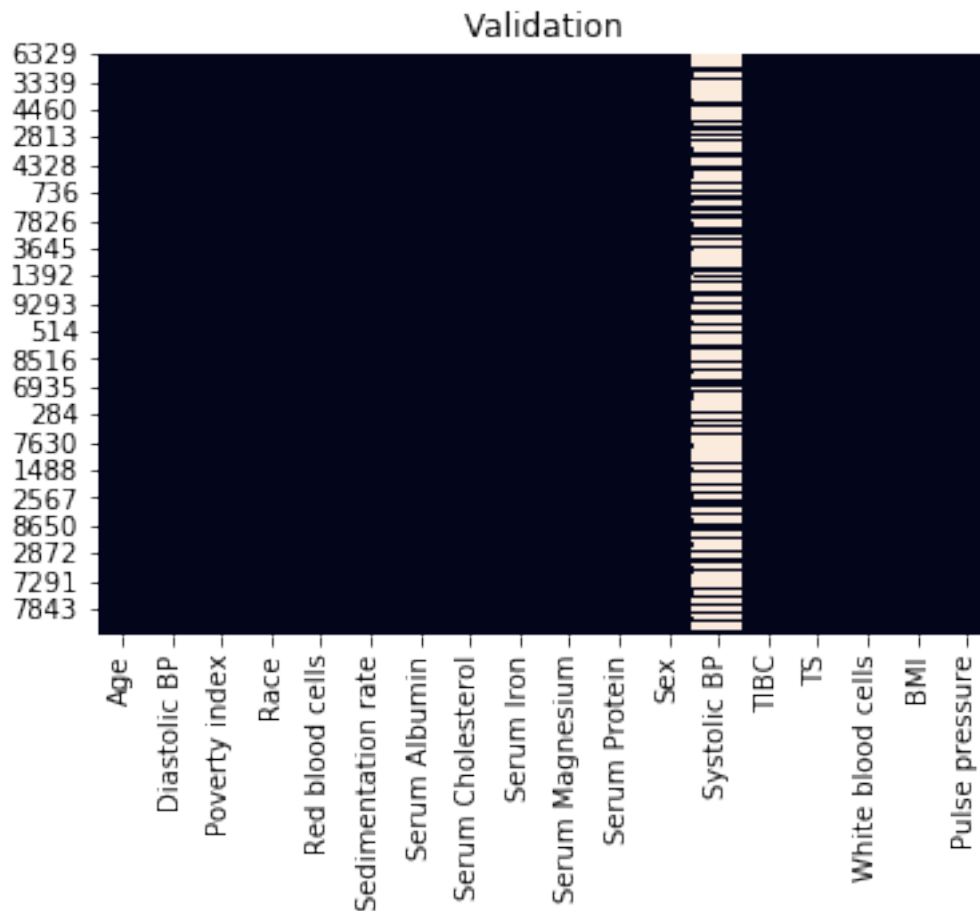
For each feature, represented as a column, values that are present are shown in black, and missing values are set in a light color.

From this plot, we can see that many values are missing for systolic blood pressure (`Systolic BP`).

Exercise 1

In the cell below, write a function to compute the fraction of cases with missing data. This will help us decide how we handle this missing data in the future.

Hints

The pandas.DataFrame.isnull() method is helpful in this case.

Use the pandas.DataFrame.any() method and set the axis parameter.

Divide the total number of rows with missing data by the total number of rows. Remember that in Python, True values are equal to 1.

```python
[8]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
     def fraction_rows_missing(df):
         '''
         Return percent of rows with any missing
         data in the dataframe.

         Input:
             df (dataframe): a pandas dataframe with potentially missing data
         Output:
             frac_missing (float): fraction of rows with missing data
         '''

         return sum(df.isnull().any(axis=1)) / len(df)
```

Test your function by running the cell below.

```python
[9]: df_test = pd.DataFrame({'a':[None, 1, 1, None], 'b':[1, None, 0, 1]})
     print("Example dataframe:\n")
     print(df_test)

     print("\nComputed fraction missing: {}, expected: {}".
      ↪format(fraction_rows_missing(df_test), 0.75))
     print(f"Fraction of rows missing from X_train: {fraction_rows_missing(X_train):.
      ↪3f}")
     print(f"Fraction of rows missing from X_val: {fraction_rows_missing(X_val):.
      ↪3f}")
     print(f"Fraction of rows missing from X_test: {fraction_rows_missing(X_test):.
      ↪3f}")
```

```
Example dataframe:

     a    b
0  NaN  1.0
1  1.0  NaN
2  1.0  0.0
3  NaN  1.0

Computed fraction missing: 0.75, expected: 0.75
Fraction of rows missing from X_train: 0.699
Fraction of rows missing from X_val: 0.704
Fraction of rows missing from X_test: 0.000
```

We see that our train and validation sets have missing values, but luckily our test set has complete cases.

As a first pass, we will begin with a **complete case analysis**, dropping all of the rows with any missing data. Run the following cell to drop these rows from our train and validation sets.

```
[10]: X_train_dropped = X_train.dropna(axis='rows')
      y_train_dropped = y_train.loc[X_train_dropped.index]
      X_val_dropped = X_val.dropna(axis='rows')
      y_val_dropped = y_val.loc[X_val_dropped.index]
```

5. Decision Trees

Having just learned about decision trees, you choose to use a decision tree classifier. Use scikit-learn to build a decision tree for the hospital dataset using the train set.

```
[11]: dt = DecisionTreeClassifier(max_depth=None, random_state=10)
      dt.fit(X_train_dropped, y_train_dropped)
```

```
[11]: DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                             max_depth=None, max_features=None, max_leaf_nodes=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=1, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, presort='deprecated',
                             random_state=10, splitter='best')
```

Next we will evaluate our model. We'll use C-Index for evaluation.

> Remember from lesson 4 of week 1 that the C-Index evaluates the ability of a model to differentiate between different classes, by quantifying how often, when considering all pairs of patients (A, B), the model says that patient A has a higher risk score than patient B when, in the observed data, patient A actually died and patient B actually lived. In our case, our model is a binary classifier, where each risk score is either 1 (the model predicts that the patient will die) or 0 (the patient will live).

> More formally, defining *permissible pairs* of patients as pairs where the outcomes are different, *concordant pairs* as permissible pairs where the patient that died had a higher risk score (i.e. our model predicted 1 for the patient that died and 0 for the one that lived), and *ties* as permissible pairs where the risk scores were equal (i.e. our model predicted 1 for both patients or 0 for both patients), the C-Index is equal to:

$$\text{C-Index} = \frac{\#\text{concordant pairs} + 0.5 \times \#\text{ties}}{\#\text{permissible pairs}}$$

Run the next cell to compute the C-Index on the train and validation set (we've given you an implementation this time).

```
[12]: y_train_preds = dt.predict_proba(X_train_dropped)[:, 1]
      print(f"Train C-Index: {cindex(y_train_dropped.values, y_train_preds)}")


      y_val_preds = dt.predict_proba(X_val_dropped)[:, 1]
      print(f"Val C-Index: {cindex(y_val_dropped.values, y_val_preds)}")
```

```
Train C-Index: 1.0
Val C-Index: 0.5629321808510638
```

Unfortunately your tree seems to be overfitting: it fits the training data so closely that it doesn't generalize well to other samples such as those from the validation set.

> The training C-index comes out to 1.0 because, when initializing `DecisionTreeClasifier`, we have left `max_depth` and `min_samples_split` unspecified. The resulting decision tree will therefore keep splitting as far as it can, which pretty much guarantees a pure fit to the training data.

To handle this, you can change some of the hyperparameters of our tree.

Exercise 2

Try and find a set of hyperparameters that improves the generalization to the validation set and recompute the C-index. If you do it right, you should get C-index above 0.6 for the validation set.

You can refer to the documentation for the sklearn DecisionTreeClassifier.

Hints

Try limiting the depth of the tree ('max_depth').

```
[13]: # Experiment with different hyperparameters for the DecisionTreeClassifier
      # until you get a c-index above 0.6 for the validation set
      dt_hyperparams = {'max_depth': 3}
```

Run the next cell to fit and evaluate the regularized tree.

```
[14]: # UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
      dt_reg = DecisionTreeClassifier(**dt_hyperparams, random_state=10)
      dt_reg.fit(X_train_dropped, y_train_dropped)

      y_train_preds = dt_reg.predict_proba(X_train_dropped)[:, 1]
      y_val_preds = dt_reg.predict_proba(X_val_dropped)[:, 1]
      print(f"Train C-Index: {cindex(y_train_dropped.values, y_train_preds)}")
      print(f"Val C-Index (expected > 0.6): {cindex(y_val_dropped.values,␣
       ↪y_val_preds)}")
```

```
Train C-Index: 0.688738755448391
Val C-Index (expected > 0.6): 0.6302692819148936
```

If you used a low `max_depth` you can print the entire tree. This allows for easy interpretability. Run the next cell to print the tree splits.

```
[15]: dot_data = StringIO()
      export_graphviz(dt_reg, feature_names=X_train_dropped.columns,␣
       ↪out_file=dot_data,
                      filled=True, rounded=True, proportion=True,␣
       ↪special_characters=True,
                      impurity=False, class_names=['neg', 'pos'], precision=2)
      graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
      Image(graph.create_png())
```

```
[15]:
```
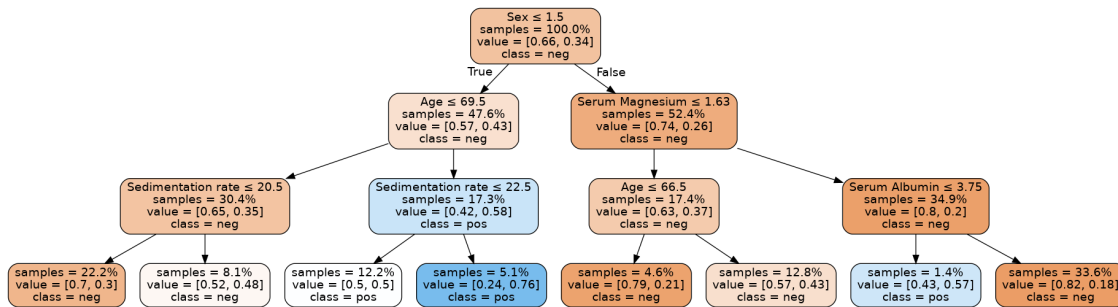
Overfitting, underfitting, and the bias-variance tradeoff

If you tested several values of max_depth, you may have seen that a value of 3 gives training and validation C-Indices of about 0.689 and 0.630, and that a max_depth of 2 gives better agreement with values of about 0.653 and 0.607. In the latter case, we have further reduced overfitting, at the cost of a minor loss in predictive performance.

Contrast this with a max_depth value of 1, which results in C-Indices of about 0.597 for the training set and 0.598 for the validation set: we have eliminated overfitting but with a much stronger degradation of predictive performance.

Lower predictive performance on the training and validation sets is indicative of the model *underfitting* the data: it neither learns enough from the training data nor is able to generalize to unseen data (the validation data in our case).

Finding a model that minimizes and acceptably balances underfitting and overfitting (e.g. selecting the model with a max_depth of 2 over the other values) is a common problem in machine learning that is known as the *bias-variance tradeoff*.

6. Random Forests

No matter how you choose hyperparameters, a single decision tree is prone to overfitting. To solve this problem, you can try **random forests**, which combine predictions from many different trees to create a robust classifier.

As before, we will use scikit-learn to build a random forest for the data. We will use the default hyperparameters.

```
[16]: rf = RandomForestClassifier(n_estimators=100, random_state=10)
      rf.fit(X_train_dropped, y_train_dropped)
```

```
[16]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                             criterion='gini', max_depth=None, max_features='auto',
                             max_leaf_nodes=None, max_samples=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
```

```
                    min_samples_leaf=1, min_samples_split=2,
                    min_weight_fraction_leaf=0.0, n_estimators=100,
                    n_jobs=None, oob_score=False, random_state=10, verbose=0,
                    warm_start=False)
```

Now compute and report the C-Index for the random forest on the training and validation set.

```
[17]: y_train_rf_preds = rf.predict_proba(X_train_dropped)[:, 1]
      print(f"Train C-Index: {cindex(y_train_dropped.values, y_train_rf_preds)}")

      y_val_rf_preds = rf.predict_proba(X_val_dropped)[:, 1]
      print(f"Val C-Index: {cindex(y_val_dropped.values, y_val_rf_preds)}")
```

```
Train C-Index: 1.0
Val C-Index: 0.6660488696808511
```

Training a random forest with the default hyperparameters results in a model that has better predictive performance than individual decision trees as in the previous section, but this model is overfitting.

We therefore need to tune (or optimize) the hyperparameters, to find a model that both has good predictive performance and minimizes overfitting.

The hyperparameters we choose to adjust will be:

- n_estimators: the number of trees used in the forest.
- max_depth: the maximum depth of each tree.
- min_samples_leaf: the minimum number (if int) or proportion (if float) of samples in a leaf.

The approach we implement to tune the hyperparameters is known as a grid search:

- We define a set of possible values for each of the target hyperparameters.

- A model is trained and evaluated for every possible combination of hyperparameters.

- The best performing set of hyperparameters is returned.

The cell below implements a hyperparameter grid search, using the C-Index to evaluate each tested model.

```
[18]: def holdout_grid_search(clf, X_train_hp, y_train_hp, X_val_hp, y_val_hp,␣
      ↪hyperparams, fixed_hyperparams={}):
          '''
          Conduct hyperparameter grid search on hold out validation set. Use holdout␣
      ↪validation.
          Hyperparameters are input as a dictionary mapping each hyperparameter name␣
      ↪to the
          range of values they should iterate over. Use the cindex function as your␣
      ↪evaluation
```

```
    function.

    Input:
        clf: sklearn classifier
        X_train_hp (dataframe): dataframe for training set input variables
        y_train_hp (dataframe): dataframe for training set targets
        X_val_hp (dataframe): dataframe for validation set input variables
        y_val_hp (dataframe): dataframe for validation set targets
        hyperparams (dict): hyperparameter dictionary mapping hyperparameter
                            names to range of values for grid search
        fixed_hyperparams (dict): dictionary of fixed hyperparameters that
                                  are not included in the grid search

    Output:
        best_estimator (sklearn classifier): fitted sklearn classifier with
→best performance on
                                              validation set
        best_hyperparams (dict): hyperparameter dictionary mapping
→hyperparameter
                                 names to values in best_estimator
    '''
    best_estimator = None
    best_hyperparams = {}

    # hold best running score
    best_score = 0.0

    # get list of param values
    lists = hyperparams.values()

    # get all param combinations
    param_combinations = list(itertools.product(*lists))
    total_param_combinations = len(param_combinations)

    # iterate through param combinations
    for i, params in enumerate(param_combinations, 1):
        # fill param dict with params
        param_dict = {}
        for param_index, param_name in enumerate(hyperparams):
            param_dict[param_name] = params[param_index]

        # create estimator with specified params
        estimator = clf(**param_dict, **fixed_hyperparams)

        # fit estimator
        estimator.fit(X_train_hp, y_train_hp)
```

```python
        # get predictions on validation set
        preds = estimator.predict_proba(X_val_hp)

        # compute cindex for predictions
        estimator_score = cindex(y_val_hp, preds[:,1])

        print(f'[{i}/{total_param_combinations}] {param_dict}')
        print(f'Val C-Index: {estimator_score}\n')

        # if new high score, update high score, best estimator
        # and best params
        if estimator_score >= best_score:
                best_score = estimator_score
                best_estimator = estimator
                best_hyperparams = param_dict

    # add fixed hyperparamters to best combination of variable hyperparameters
    best_hyperparams.update(fixed_hyperparams)

    return best_estimator, best_hyperparams
```

Exercise 3

In the cell below, define the values you want to run the hyperparameter grid search on, and run the cell to find the best-performing set of hyperparameters.

Your objective is to get a C-Index above 0.6 on both the train and validation set.

Hints

n_estimators: try values greater than 100

max_depth: try values in the range 1 to 100

min_samples_leaf: try float values below .5 and/or int values greater than 2

```python
[19]: def random_forest_grid_search(X_train_dropped, y_train_dropped, X_val_dropped,␣
    ↪y_val_dropped):

    # Define ranges for the chosen random forest hyperparameters
    hyperparams = {'n_estimators': [100,150,200], 'max_depth': [3,4,5],␣
    ↪'min_samples_leaf': [3,4]}



    fixed_hyperparams = {
        'random_state': 10,
    }


    rf = RandomForestClassifier
```

```python
    best_rf, best_hyperparams = holdout_grid_search(rf, X_train_dropped,
 →y_train_dropped,

                                                    X_val_dropped,
 →y_val_dropped, hyperparams,

                                                    fixed_hyperparams)

    print(f"Best hyperparameters:\n{best_hyperparams}")


    y_train_best = best_rf.predict_proba(X_train_dropped)[:, 1]
    print(f"Train C-Index: {cindex(y_train_dropped, y_train_best)}")

    y_val_best = best_rf.predict_proba(X_val_dropped)[:, 1]
    print(f"Val C-Index: {cindex(y_val_dropped, y_val_best)}")

    # add fixed hyperparamters to best combination of variable hyperparameters
    best_hyperparams.update(fixed_hyperparams)

    return best_rf, best_hyperparams
```

```python
[20]: best_rf, best_hyperparams = random_forest_grid_search(X_train_dropped,
 →y_train_dropped, X_val_dropped, y_val_dropped)
```

```
[1/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.6772273936170212

[2/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.6771110372340425

[3/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.6712599734042554

[4/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.6732047872340425

[5/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.6697972074468085

[6/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.6693317819148936

[7/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.6797207446808511

[8/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.6803523936170213
```

```
[9/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.6743184840425532

[10/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.6769614361702128

[11/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.673936170212766

[12/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.6719082446808511

[13/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.6809175531914894

[14/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.6812832446808511

[15/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.6752659574468085

[16/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.6782912234042553

[17/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.6745844414893617

[18/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.6736868351063829

Best hyperparameters:
{'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 4, 'random_state': 10}
Train C-Index: 0.7791152740424743
Val C-Index: 0.6812832446808511
```

Finally, evaluate the model on the test set. This is a crucial step, as trying out many combinations of hyperparameters and evaluating them on the validation set could result in a model that ends up overfitting the validation set. We therefore need to check if the model performs well on unseen data, which is the role of the test set, which we have held out until now.

```
[21]:   # UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        y_test_best = best_rf.predict_proba(X_test)[:, 1]

        print(f"Test C-Index: {cindex(y_test.values, y_test_best)}")
```

```
Test C-Index: 0.7000063288468841
```

Your C-Index on the test set should be greater than 0.6.

7. Imputation

You've now built and optimized a random forest model on our data. However, there was still a drop in test C-Index. This might be because you threw away more than half of the data of our data because of missing values for systolic blood pressure. Instead, we can try filling in, or imputing, these values.

First, let's explore to see if our data is missing at random or not. Let's plot histograms of the dropped rows against each of the covariates (aside from systolic blood pressure) to see if there is a trend. Compare these to the histograms of the feature in the entire dataset. Try to see if one of the covariates has a signficantly different distribution in the two subsets.

```
[22]: dropped_rows = X_train[X_train.isnull().any(axis=1)]

columns_except_Systolic_BP = [col for col in X_train.columns if col not in
 ↪['Systolic BP']]

for col in columns_except_Systolic_BP:
    sns.distplot(X_train.loc[:, col], norm_hist=True, kde=False, label='full
 ↪data')
    sns.distplot(dropped_rows.loc[:, col], norm_hist=True, kde=False,
 ↪label='without missing data')
    plt.legend()

    plt.show()
```
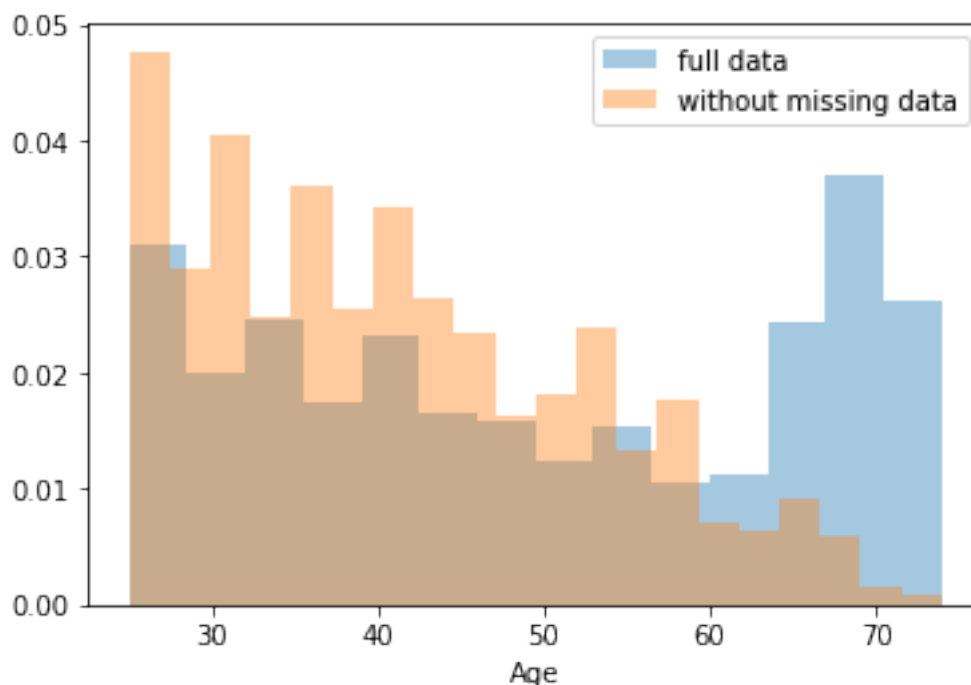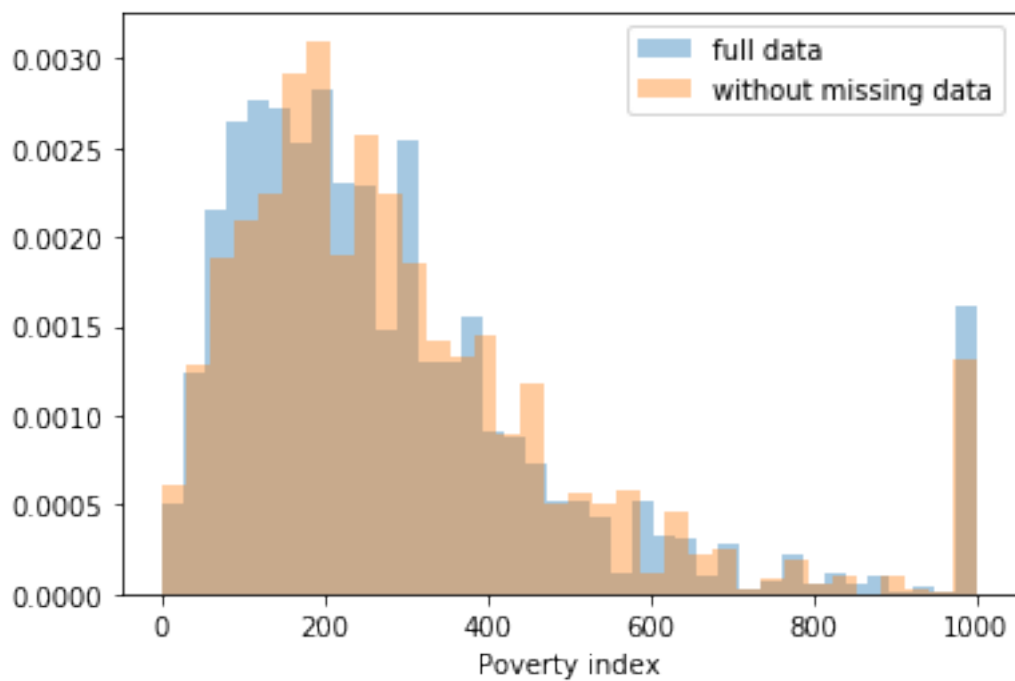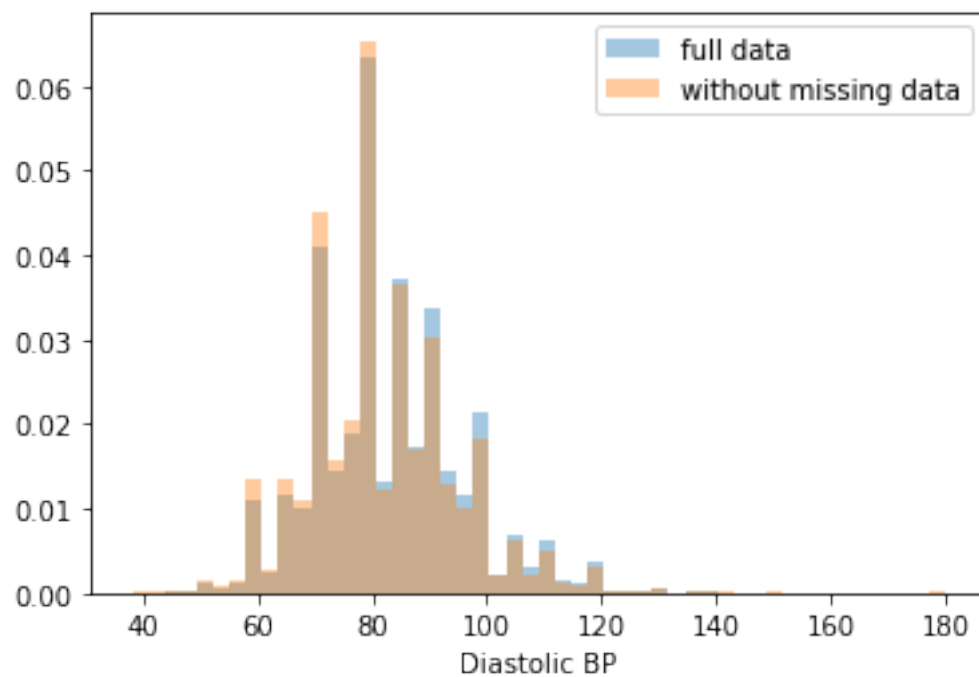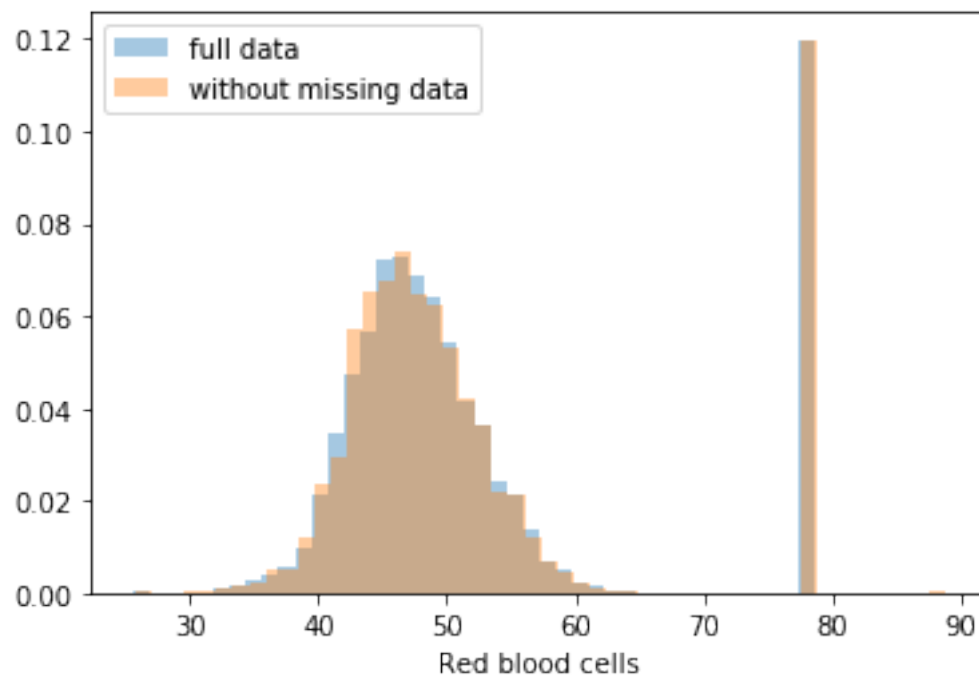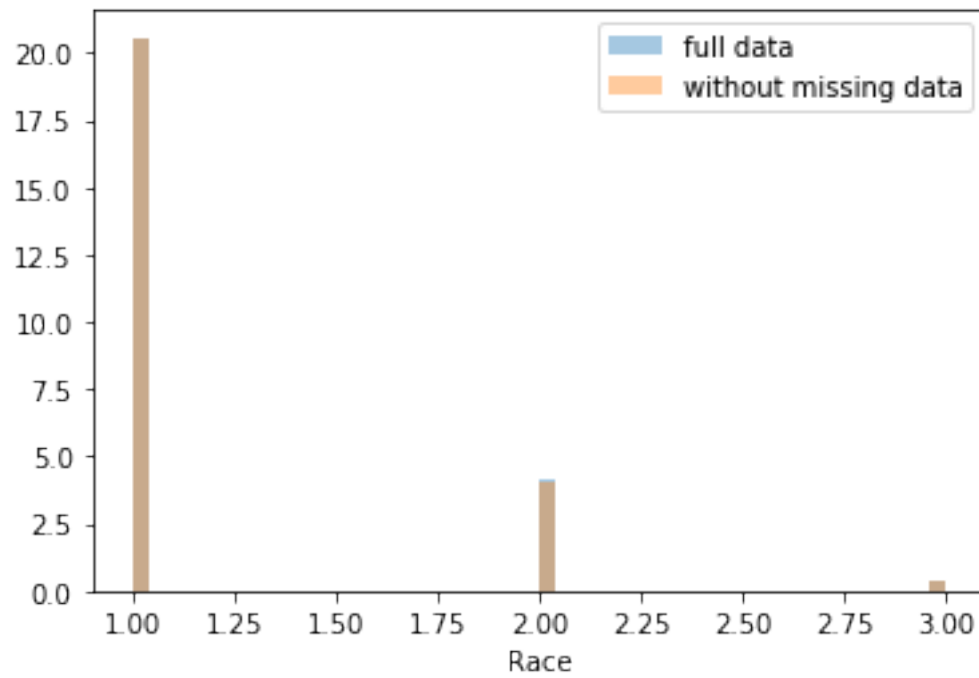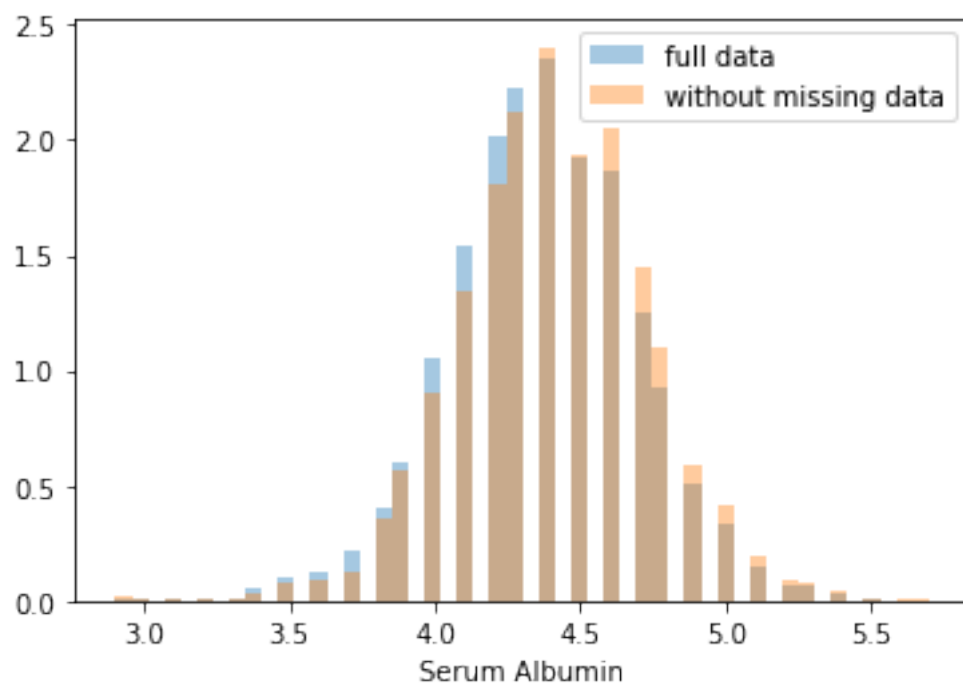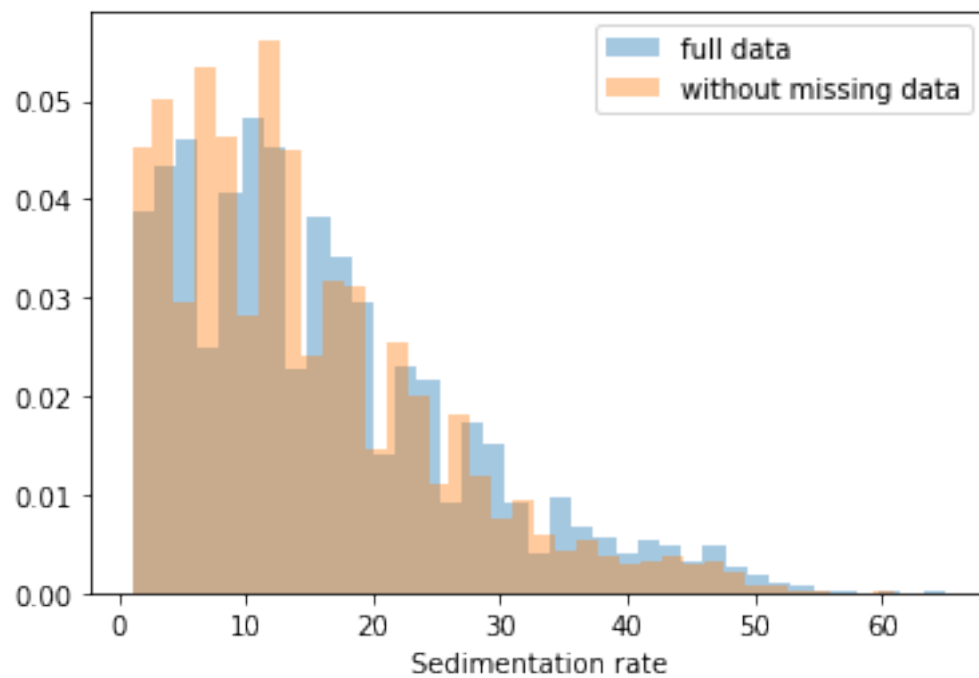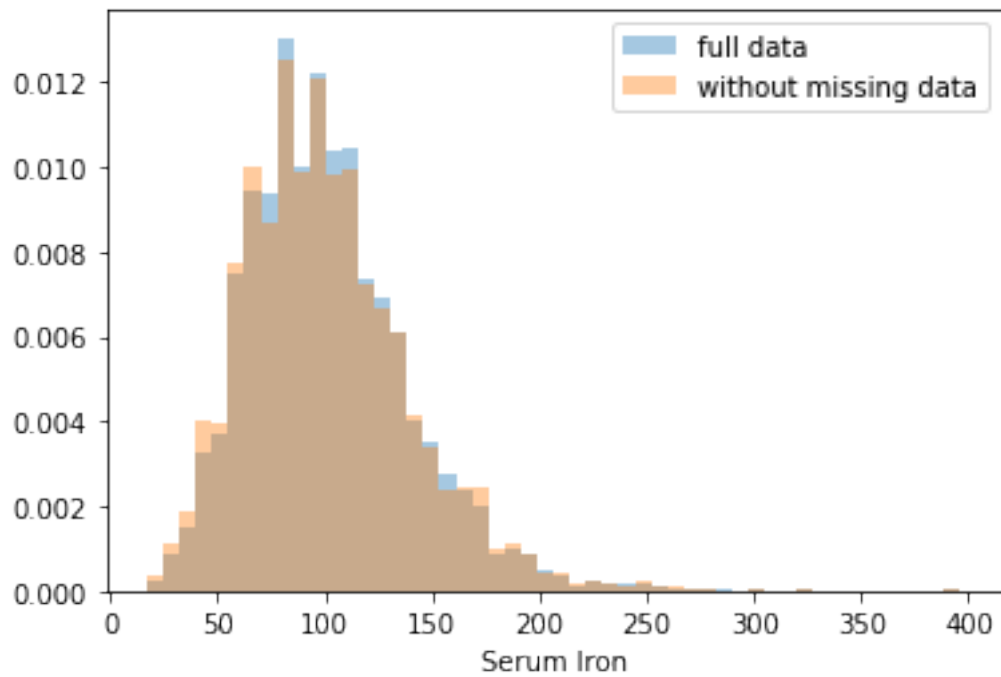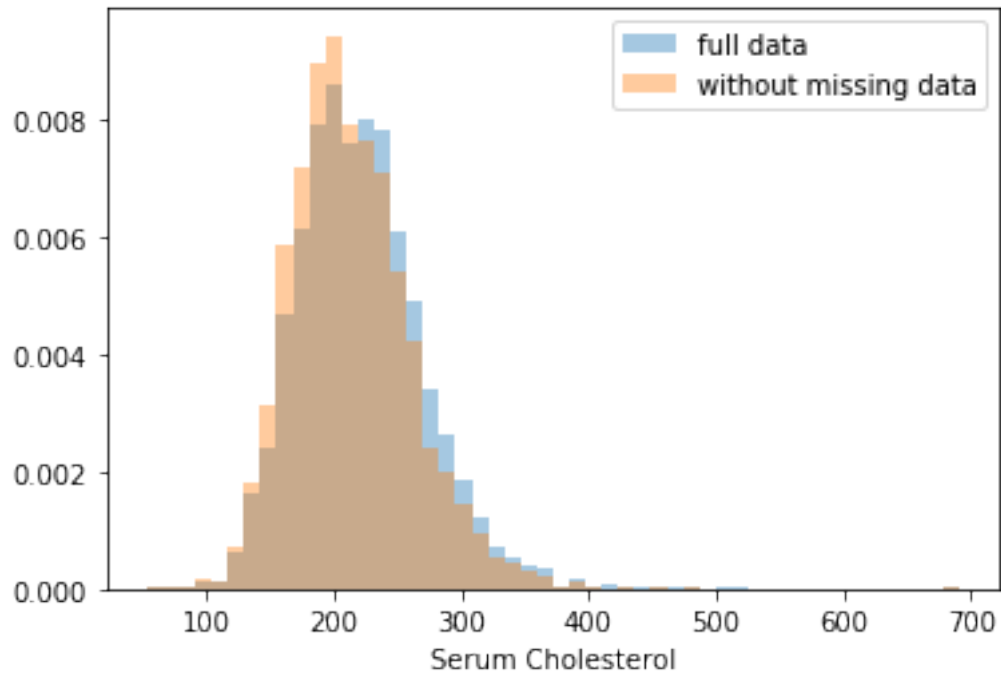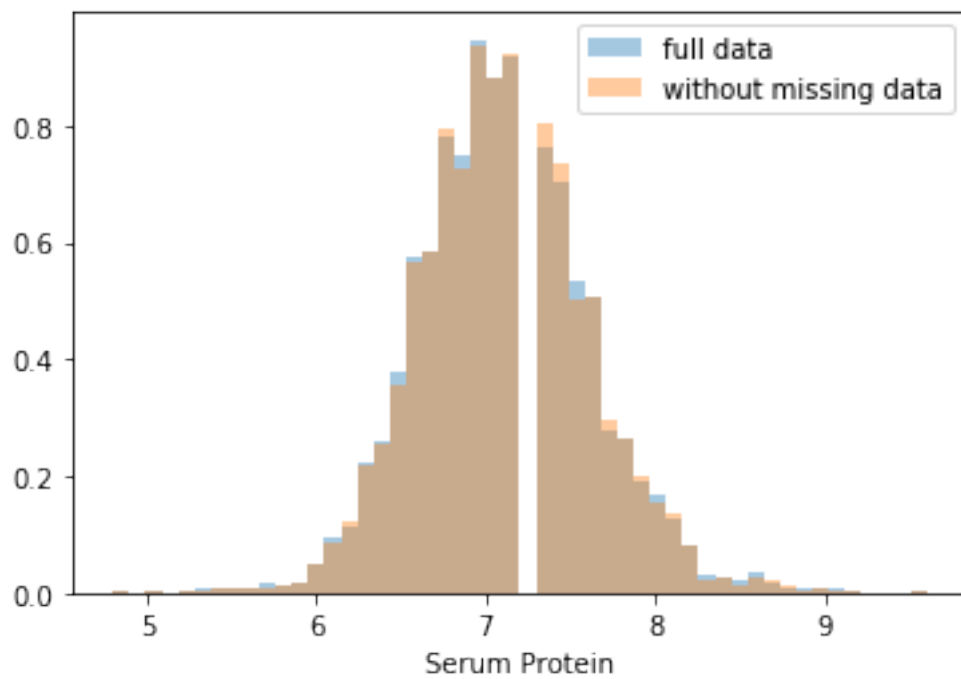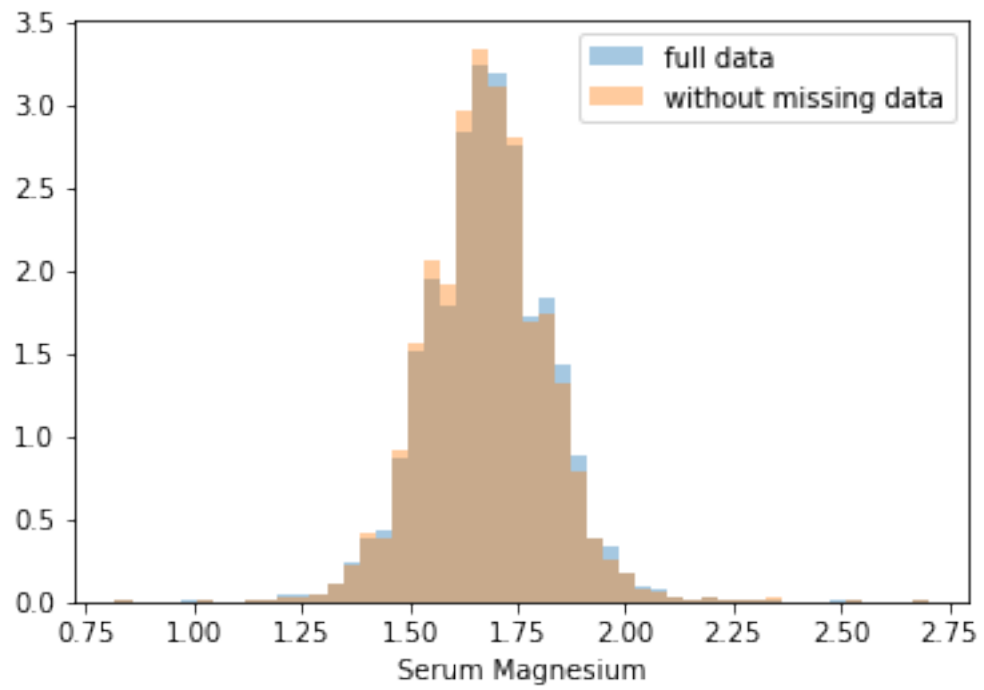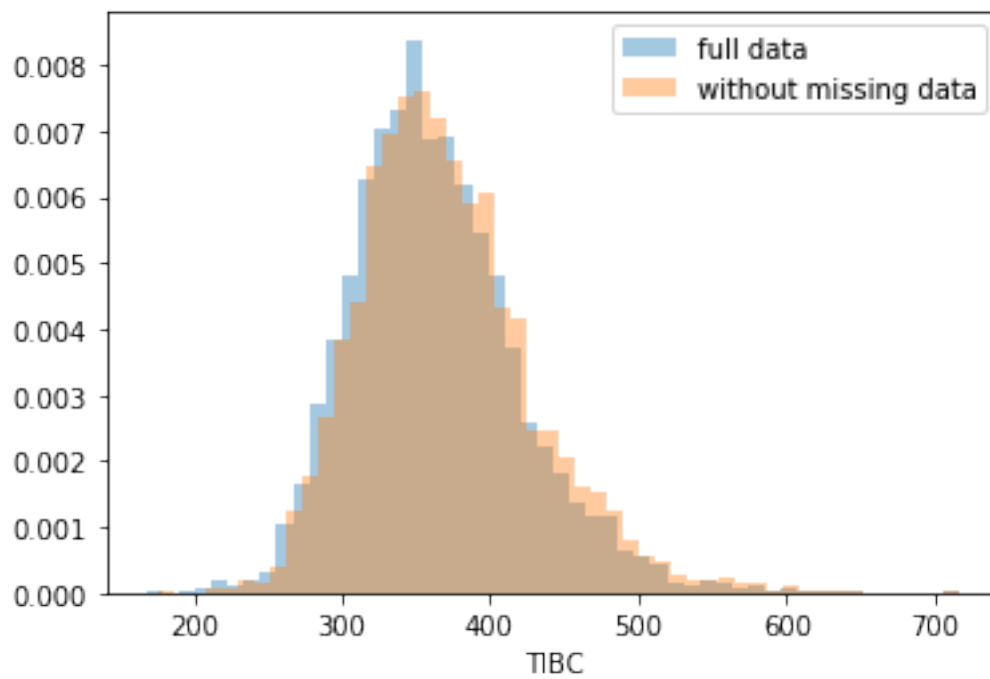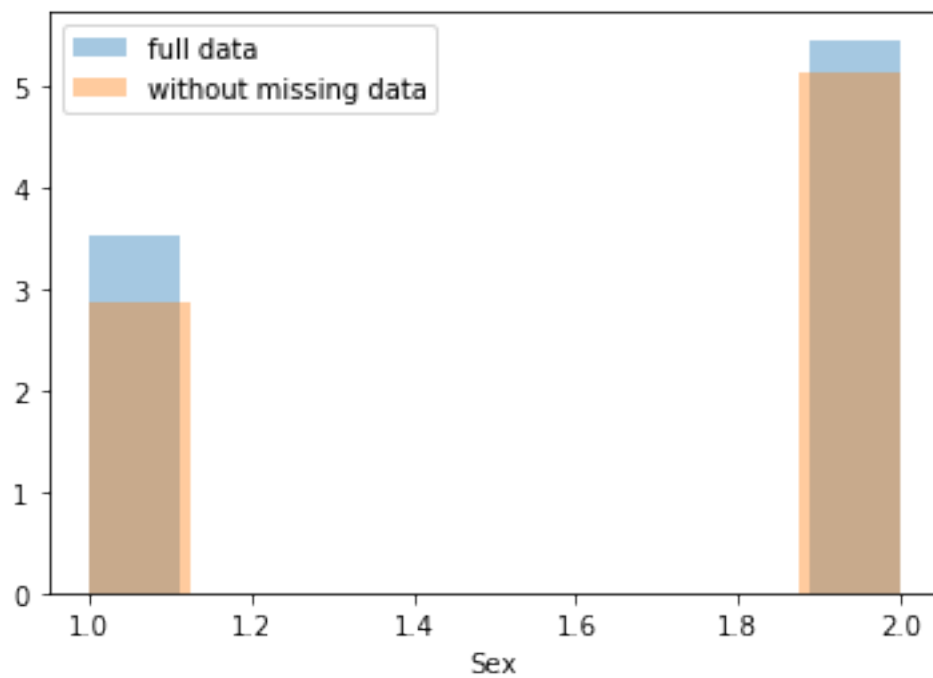
Most of the covariates are distributed similarly whether or not we have discarded rows with missing data. In other words missingness of the data is independent of

these covariates.

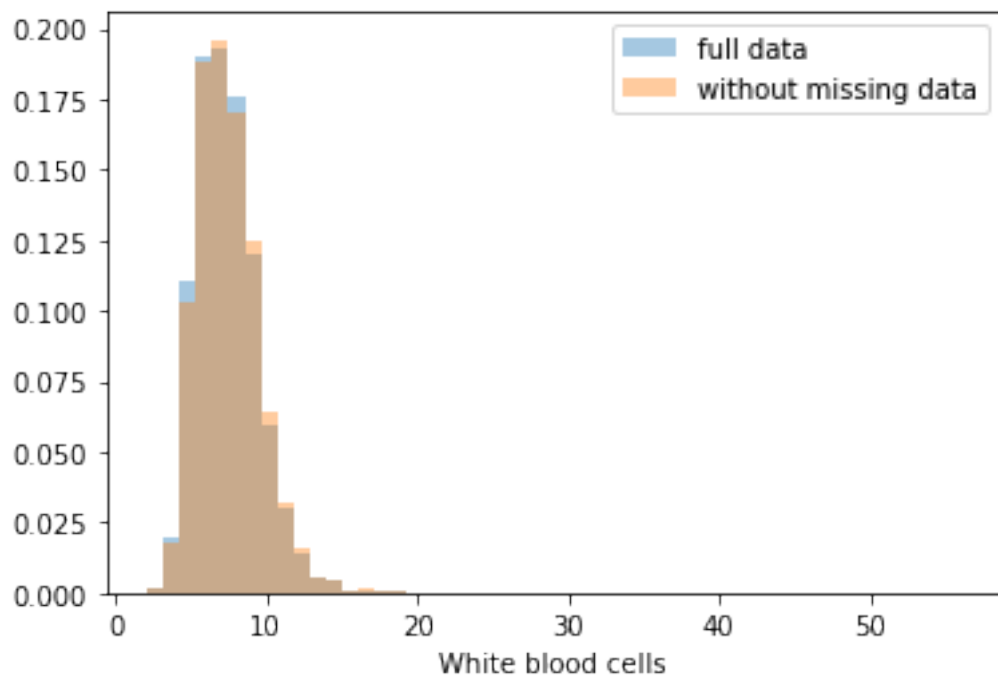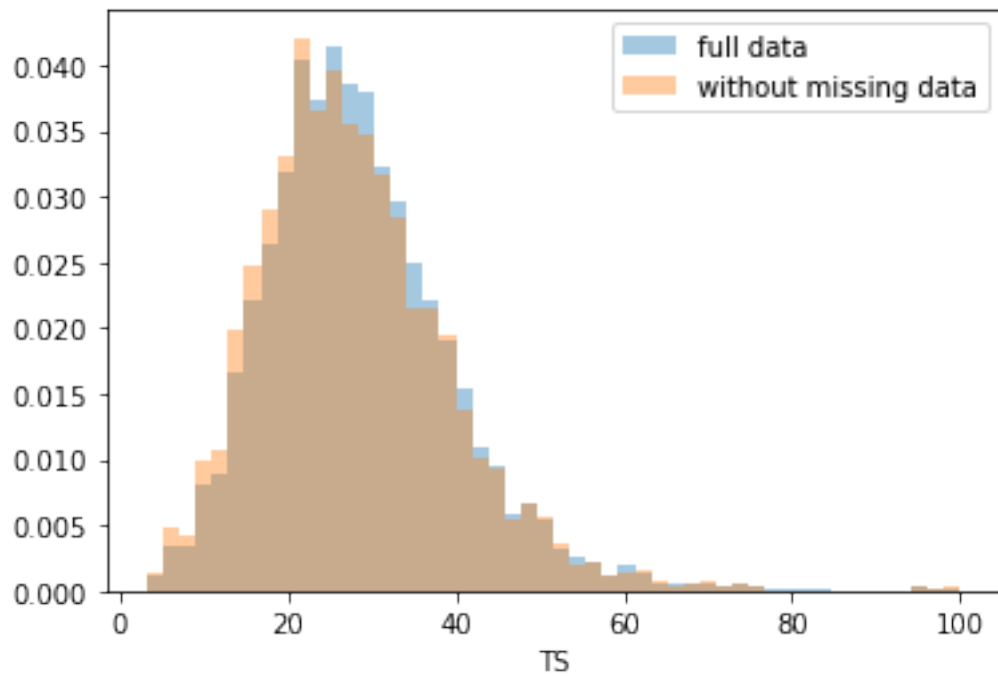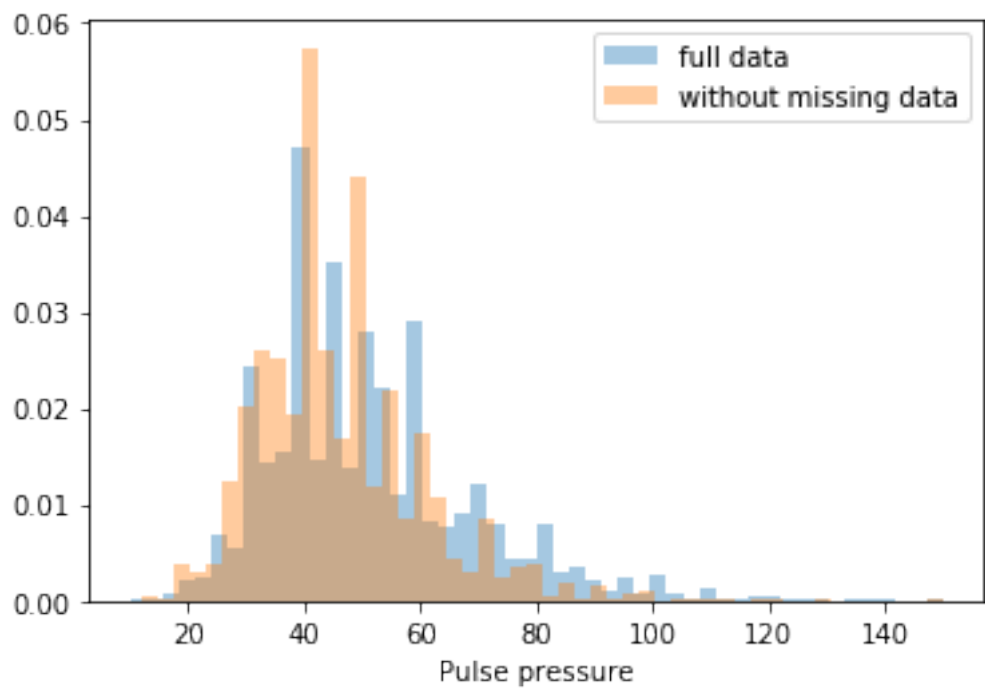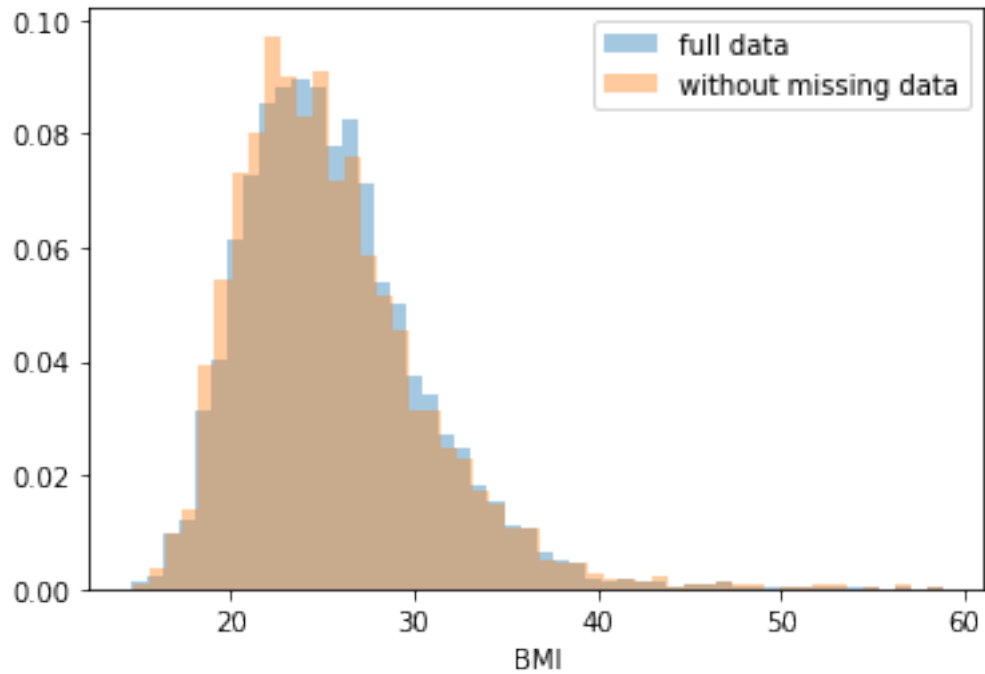If this had been true across *all* covariates, then the data would have been said to be **missing completely at random (MCAR)**.

But when considering the age covariate, we see that much more data tends to be missing for patients over 65. The reason could be that blood pressure was measured less frequently for old people to avoid placing additional burden on them.

As missingness is related to one or more covariates, the missing data is said to be **missing at random (MAR)**.

Based on the information we have, there is however no reason to believe that the *values* of the missing data --- or specifically the values of the missing systolic blood pressures --- are related to the age of the patients. If this was the case, then this data would be said to be **missing not at random (MNAR)**.

8. Error Analysis

Exercise 4 Using the information from the plots above, try to find a subgroup of the test data on which the model performs poorly. You should be able to easily find a subgroup of at least 250 cases on which the model has a C-Index of less than 0.69.

Hints

Define a mask using a feature and a threshold, e.g. patients with a BMI below 20: mask = X_test[`BMI'] < 20 .

Try to find a subgroup for which the model had little data.

```
[23]:  # UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
       def bad_subset(forest, X_test, y_test):
           # define mask to select large subset with poor performance
           # currently mask defines the entire set

           ### START CODE HERE (REPLACE the code after 'mask =' with your code) ###
           mask = X_test.Age < 40
           ### END CODE HERE ###

           X_subgroup = X_test[mask]
           y_subgroup = y_test[mask]
           subgroup_size = len(X_subgroup)

           y_subgroup_preds = forest.predict_proba(X_subgroup)[:, 1]
           performance = cindex(y_subgroup.values, y_subgroup_preds)

           return performance, subgroup_size
```

Test Your Work

```
[24]: performance, subgroup_size = bad_subset(best_rf, X_test, y_test)
      print("Subgroup size should greater than 250, performance should be less than 0.
       →69 ")
      print(f"Subgroup size: {subgroup_size}, C-Index: {performance}")
```

Subgroup size should greater than 250, performance should be less than 0.69
Subgroup size: 586, C-Index: 0.6274714828897339

Expected **Output** Note, your actual output will vary depending on the hyper-parameters that you chose and the mask that you chose. – Make sure that the c-index is less than 0.69

Subgroup size: 586, C-Index: 0.6275

Bonus: – See if you can get a c-index as low as 0.53

Subgroup size: 251, C-Index: 0.5331

9. Imputation Approaches

Seeing that our data is not missing completely at random, we can handle the missing values by replacing them with substituted values based on the other values that we have. This is known as imputation.

The first imputation strategy that we will use is **mean substitution**: we will replace the missing values for each feature with the mean of the available values. In the next cell, use the SimpleImputer from sklearn to use mean imputation for the missing values.

```
[25]: # Impute values using the mean
      imputer = SimpleImputer(strategy='mean')
      imputer.fit(X_train)
      X_train_mean_imputed = pd.DataFrame(imputer.transform(X_train), columns=X_train.
       →columns)
      X_val_mean_imputed = pd.DataFrame(imputer.transform(X_val), columns=X_val.
       →columns)
```

Exercise 5 Now perform a hyperparameter grid search to find the best-performing random forest model, and report results on the test set.

Define the parameter ranges for the hyperparameter search in the next cell, and run the cell.

Target performance Make your test c-index at least 0.74 or higher

Hints

n_estimators: try values greater than 100

max_depth: try values in the range 1 to 100

min_samples_leaf: try float values below .5 and/or int values greater than 2

```
[26]: # Define ranges for the random forest hyperparameter search
      hyperparams = {'n_estimators': [100,150,200], 'max_depth': [3,4,5],
      ↪'min_samples_leaf': [3,4]}
```

```
[27]: # UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
      rf = RandomForestClassifier

      rf_mean_imputed, best_hyperparams_mean_imputed = holdout_grid_search(rf,
      ↪X_train_mean_imputed, y_train,

                                                                          ␣
      ↪X_val_mean_imputed, y_val,

                                                                          ␣
      ↪hyperparams, {'random_state': 10})

      print("Performance for best hyperparameters:")

      y_train_best = rf_mean_imputed.predict_proba(X_train_mean_imputed)[:, 1]
      print(f"- Train C-Index: {cindex(y_train, y_train_best):.4f}")

      y_val_best = rf_mean_imputed.predict_proba(X_val_mean_imputed)[:, 1]
      print(f"- Val C-Index: {cindex(y_val, y_val_best):.4f}")

      y_test_imp = rf_mean_imputed.predict_proba(X_test)[:, 1]
      print(f"- Test C-Index: {cindex(y_test, y_test_imp):.4f}")
```

```
[1/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.7355381411780544

[2/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.7351896364911549

[3/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.7433713540004212

[4/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7431542171238483

[5/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7494088448535303

[6/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7476066087779754

[7/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.737671510990383

[8/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 4}
```

```
Val C-Index: 0.7375510000238851

[9/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.745231131348268

[10/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7450291940530552

[11/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7483622451084491

[12/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7477325481663877

[13/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.7396604847797906

[14/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.7393901493684574

[15/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.745559008031893

[16/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7454830101250925

[17/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7495499838233027

[18/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7489767424691502

Performance for best hyperparameters:
- Train C-Index: 0.8109
- Val C-Index: 0.7495
- Test C-Index: 0.7805
```

Expected   output Note, your actual c-index values will vary depending on the hyper-parameters that you choose.
- Try to get a good Test c-index, similar these numbers below:

```
Performance for best hyperparameters:
- Train C-Index: 0.8109
- Val C-Index: 0.7495
- Test C-Index: 0.7805
```

Next, we will apply another imputation strategy, known as **multivariate feature imputation**, using scikit-learn's IterativeImputer class (see the documentation).

With this strategy, for each feature that is missing values, a regression model is trained to predict observed values based on all of the other features, and the missing values are inferred using this model. As a single iteration across all features may not be enough to impute all missing values, several iterations may be performed, hence the name of the class IterativeImputer.

In the next cell, use IterativeImputer to perform multivariate feature imputation.

> Note that the first time the cell is run, imputer.fit(X_train) may fail with the message LinAlgError: SVD did not converge: simply re-run the cell.

```
[33]:  # Impute using regression on other covariates
       imputer = IterativeImputer(random_state=0, sample_posterior=False, max_iter=1,
       →min_value=0)
       imputer.fit(X_train)
       X_train_imputed = pd.DataFrame(imputer.transform(X_train), columns=X_train.
       →columns)
       X_val_imputed = pd.DataFrame(imputer.transform(X_val), columns=X_val.columns)
```

Exercise 6

Perform a hyperparameter grid search to find the best-performing random forest model, and report results on the test set. Define the parameter ranges for the hyperparameter search in the next cell, and run the cell.

Target performance Try to get a text c-index of at least 0.74 or higher.

Hints

n_estimators: try values greater than 100

max_depth: try values in the range 1 to 100

min_samples_leaf: try float values below .5 and/or int values greater than 2

```
[34]:  # Define ranges for the random forest hyperparameter search
       hyperparams = {'n_estimators': [100,150,200], 'max_depth': [3,4,5],
       →'min_samples_leaf': [3,4]}
```

```
[35]:  # UNQ_C6 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
       rf = RandomForestClassifier

       rf_imputed, best_hyperparams_imputed = holdout_grid_search(rf, X_train_imputed,
       →y_train,
                                                                  X_val_imputed, y_val,
                                                                  hyperparams,
       →{'random_state': 10})
```

```python
print("Performance for best hyperparameters:")

y_train_best = rf_imputed.predict_proba(X_train_imputed)[:, 1]
print(f"- Train C-Index: {cindex(y_train, y_train_best):.4f}")

y_val_best = rf_imputed.predict_proba(X_val_imputed)[:, 1]
print(f"- Val C-Index: {cindex(y_val, y_val_best):.4f}")

y_test_imp = rf_imputed.predict_proba(X_test)[:, 1]
print(f"- Test C-Index: {cindex(y_test, y_test_imp):.4f}")
```

[1/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.7329770117188772

[2/18] {'n_estimators': 100, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.7325264526999885

[3/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.7406224011430085

[4/18] {'n_estimators': 100, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7401512141208454

[5/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7439022536636419

[6/18] {'n_estimators': 100, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7433290123094896

[7/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.7338140743780657

[8/18] {'n_estimators': 150, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.7336707640395276

[9/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.7409926195175653

[10/18] {'n_estimators': 150, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7403889790006927

[11/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7430380488948819

[12/18] {'n_estimators': 150, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7422932694082369

```
[13/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 3}
Val C-Index: 0.7356792801478268

[14/18] {'n_estimators': 200, 'max_depth': 3, 'min_samples_leaf': 4}
Val C-Index: 0.735444772321128

[15/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 3}
Val C-Index: 0.7429316518253611

[16/18] {'n_estimators': 200, 'max_depth': 4, 'min_samples_leaf': 4}
Val C-Index: 0.7425451481850615

[17/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 3}
Val C-Index: 0.7453787844243376

[18/18] {'n_estimators': 200, 'max_depth': 5, 'min_samples_leaf': 4}
Val C-Index: 0.7451247342787473


Performance for best hyperparameters:
- Train C-Index: 0.8131
- Val C-Index: 0.7454
- Test C-Index: 0.7797
```

Expected Output Note, your actual output will vary depending on the hyper-parameters that you chose and the mask that you chose.

```
Performance for best hyperparameters:
- Train C-Index: 0.8131
- Val C-Index: 0.7454
- Test C-Index: 0.7797
```

10. Comparison

For good measure, retest on the subgroup from before to see if your new models do better.

```
[36]: performance, subgroup_size = bad_subset(best_rf, X_test, y_test)
      print(f"C-Index (no imputation): {performance}")

      performance, subgroup_size = bad_subset(rf_mean_imputed, X_test, y_test)
      print(f"C-Index (mean imputation): {performance}")

      performance, subgroup_size = bad_subset(rf_imputed, X_test, y_test)
      print(f"C-Index (multivariate feature imputation): {performance}")
```

```
C-Index (no imputation): 0.6274714828897339
C-Index (mean imputation): 0.5974334600760456
C-Index (multivariate feature imputation): 0.5903992395437262
```

We should see that avoiding complete case analysis (i.e. analysis only on observations for which there is no missing data) allows our model to generalize a bit better. Remember to examine your missing cases to judge whether they are missing at random or not!

11. Explanations: SHAP

Using a random forest has improved results, but we've lost some of the natural interpretability of trees. In this section we'll try to explain the predictions using slightly more sophisticated techniques.

You choose to apply **SHAP (SHapley Additive exPlanations)** , a cutting edge method that explains predictions made by black-box machine learning models (i.e. models which are too complex to be understandable by humans as is).

> Given a prediction made by a machine learning model, SHAP values explain the prediction by quantifying the additive importance of each feature to the prediction. SHAP values have their roots in cooperative game theory, where Shapley values are used to quantify the contribution of each player to the game.
>
> Although it is computationally expensive to compute SHAP values for general black-box models, in the case of trees and forests there exists a fast polynomial-time algorithm. For more details, see the TreeShap paper.

We'll use the shap library to do this for our random forest model. Run the next cell to output the most at risk individuals in the test set according to our model.

```
[37]: X_test_risk = X_test.copy(deep=True)
      X_test_risk.loc[:, 'risk'] = rf_imputed.predict_proba(X_test_risk)[:, 1]
      X_test_risk = X_test_risk.sort_values(by='risk', ascending=False)
      X_test_risk.head()
```

[37]:
	Age	Diastolic BP	Poverty index	Race	Red blood cells \
5493	67.0	80.0	30.0	1.0	77.7
1017	65.0	98.0	16.0	1.0	49.4
2050	66.0	100.0	69.0	2.0	42.9
6337	69.0	80.0	233.0	1.0	77.7
2608	71.0	80.0	104.0	1.0	43.8

	Sedimentation rate	Serum Albumin	Serum Cholesterol	Serum Iron \
5493	59.0	3.4	231.0	36.0
1017	30.0	3.4	124.0	129.0
2050	47.0	3.8	233.0	170.0
6337	48.0	4.2	159.0	87.0
2608	23.0	4.0	201.0	119.0

	Serum Magnesium	Serum Protein	Sex	Systolic BP	TIBC	TS \
5493	1.40	6.3	1.0	170.0	202.0	17.8

1017	1.59	7.7	1.0	184.0	293.0	44.0
2050	1.42	8.6	1.0	180.0	411.0	41.4
6337	1.81	6.9	1.0	146.0	291.0	29.9
2608	1.60	7.0	1.0	166.0	311.0	38.3

	White blood cells	BMI	Pulse pressure	risk
5493	8.4	17.029470	90.0	0.619022
1017	5.9	30.858853	86.0	0.545443
2050	7.2	22.129498	80.0	0.527768
6337	15.2	17.931276	66.0	0.526019
2608	6.3	17.760766	86.0	0.525624

We can use SHAP values to try and understand the model output on specific individuals using force plots. Run the cell below to see a force plot on the riskiest individual.

```
[38]: explainer = shap.TreeExplainer(rf_imputed)
      i = 0
      shap_value = explainer.shap_values(X_test.loc[X_test_risk.index[i], :])[1]
      shap.force_plot(explainer.expected_value[1], shap_value, feature_names=X_test.
       →columns, matplotlib=True)
```
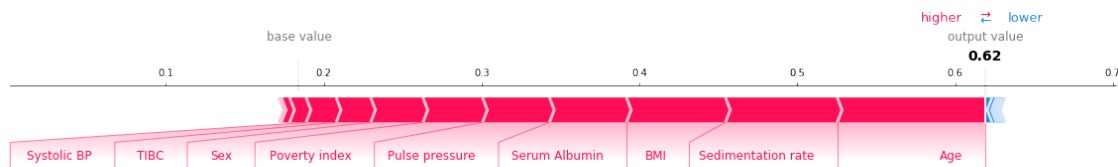

How to read this chart: – The red sections on the left are features which push the model towards the final prediction in the positive direction (i.e. a higher Age increases the predicted risk). – The blue sections on the right are features that push the model towards the final prediction in the negative direction (if an increase in a feature leads to a lower risk, it will be shown in blue). – Note that the exact output of your chart will differ depending on the hyper-parameters that you choose for your model.

We can also use SHAP values to understand the model output in aggregate. Run the next cell to initialize the SHAP values (this may take a few minutes).

```
[39]: shap_values = shap.TreeExplainer(rf_imputed).shap_values(X_test)[1]
```

Run the next cell to see a summary plot of the SHAP values for each feature on each of the test examples. The colors indicate the value of the feature.

```
[40]:  shap.summary_plot(shap_values, X_test)
```
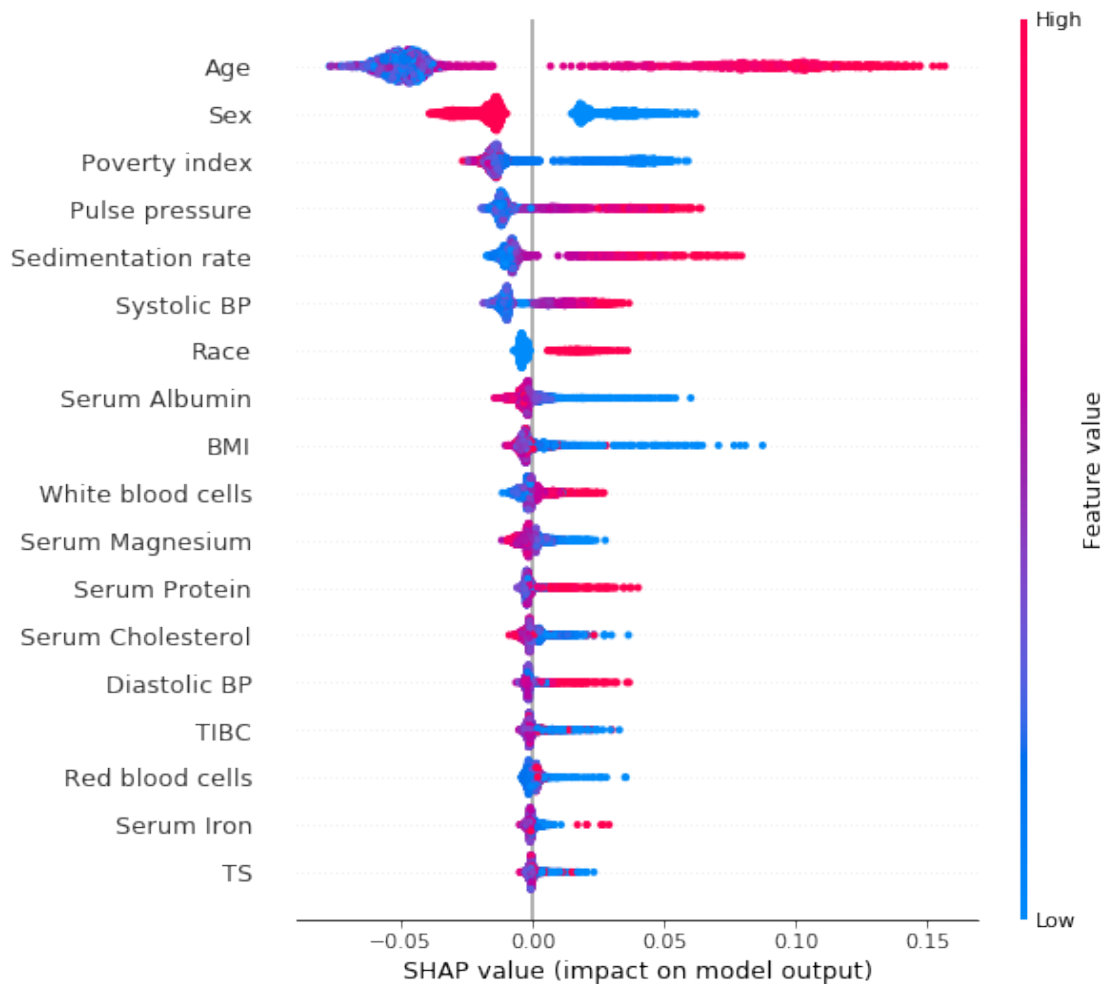

Clearly we see that being a woman (sex = 2.0, as opposed to men for which sex = 1.0) has a negative SHAP value, meaning that it reduces the risk of dying within 10 years. High age and high systolic blood pressure have positive SHAP values, and are therefore related to increased mortality.

You can see how features interact using dependence plots. These plot the SHAP value for a given feature for each data point, and color the points in using the value for another feature. This lets us begin to explain the variation in SHAP value for a single value of the main feature.

Run the next cell to see the interaction between Age and Sex.

```
[41]:  shap.dependence_plot('Age', shap_values, X_test, interaction_index='Sex')
```
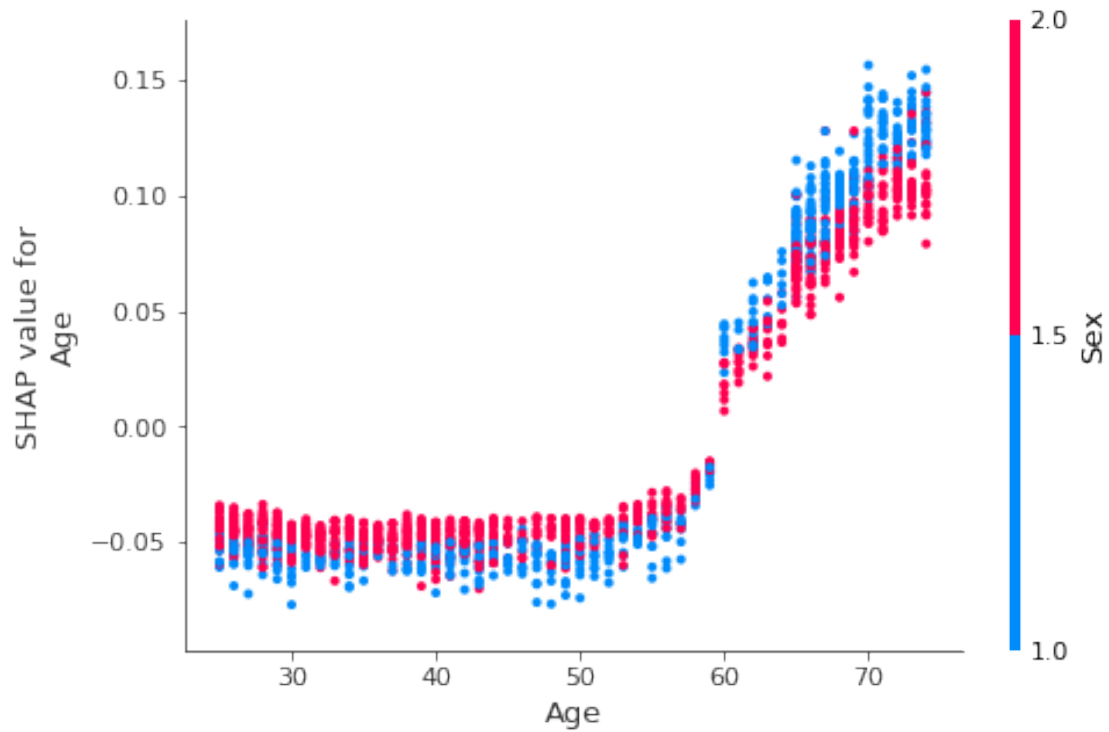
We see that while Age > 50 is generally bad (positive SHAP value), being a woman generally reduces the impact of age. This makes sense since we know that women generally live longer than men.

Let's now look at poverty index and age.

```
[42]: shap.dependence_plot('Poverty index', shap_values, X_test,
      ↪interaction_index='Age')
```
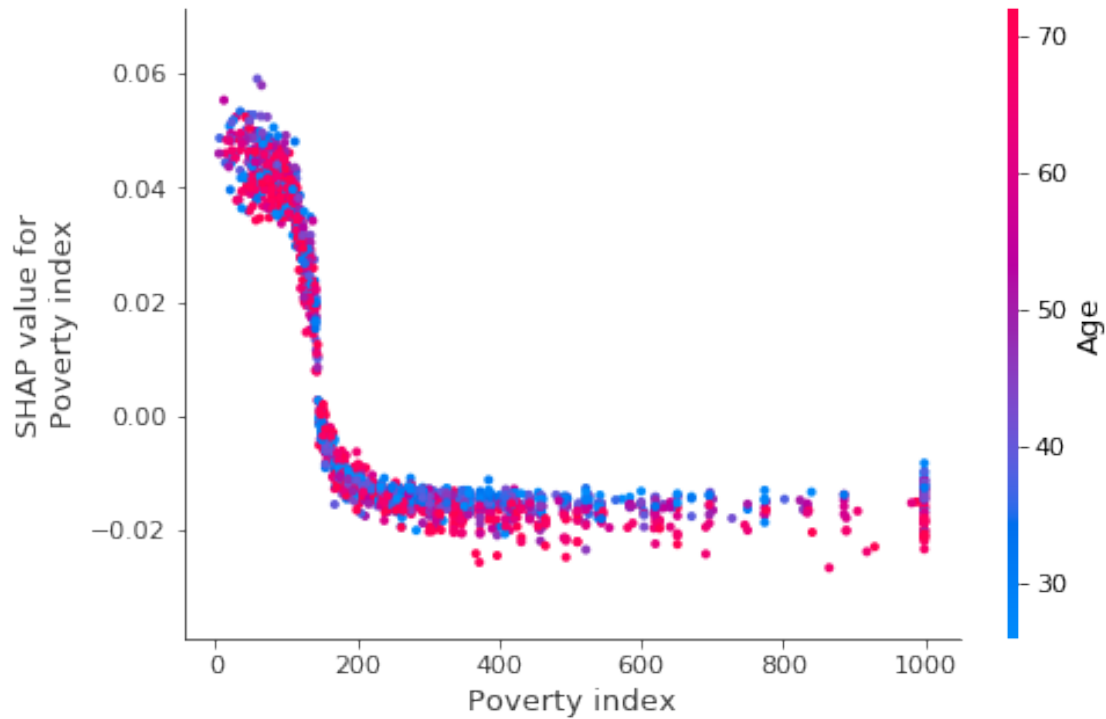
We see that the impact of poverty index drops off quickly, and for higher income individuals age begins to explain much of variation in the impact of poverty index.

Try some other pairs and see what other interesting relationships you can find!

2   Congratulations!

You have completed the second assignment in Course 2. Along the way you've learned to fit decision trees, random forests, and deal with missing data. Now you're ready to move on to week 3!