CSC 4850 Machine Learning Final Project

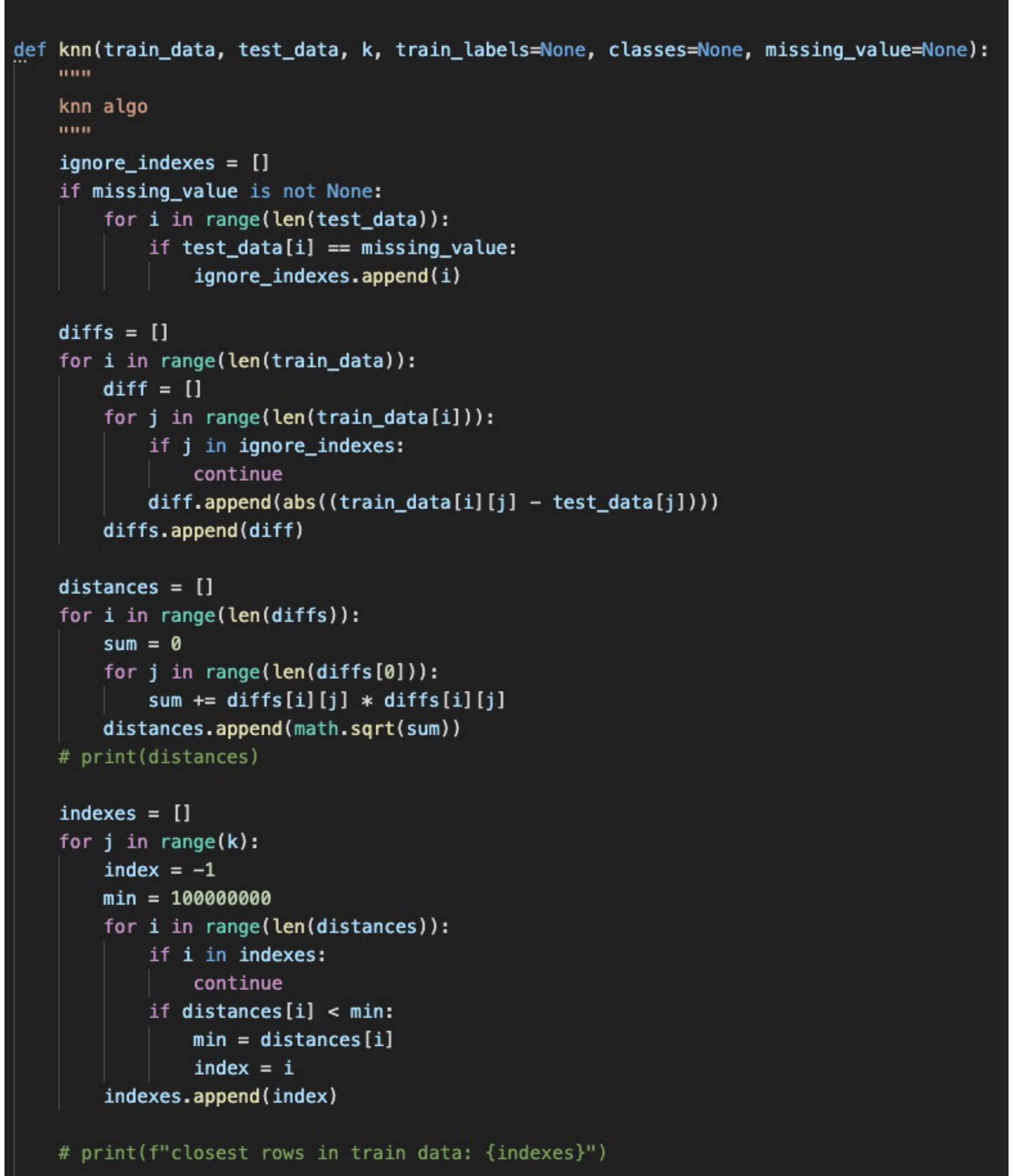
Jack Gordon

Part 1: Classification

- To classify the data, an implementation of the k-nearest neighbors algorithm was created using Python and the NumPy library.
- In the function *knn*, the required parameters are the training data, a test input, and the k variable. Optional parameters include training labels for the train data and the classes of the training label, so that predictions can be made.
- A third optional parameter is a missing value parameter, which enables the algorithm to skip over any columns in the test data that contain this value.

Part 1: Classification

- Between every row in the training data and the test row, an array of Euclidean distances is calculated.
- Then for the k smallest numbers in this distance array, the corresponding indexes of the training rows are found.
- These k nearest neighbors are returned.
- If training labels and classes are given, then the function can additionally return the prediction of the class label based on the most frequently occurring class in the nearest neighbors.
- An additional function called *knn_learn* is defined which loops the *knn* function over a set of testing data, so that a list of predictions can be returned.



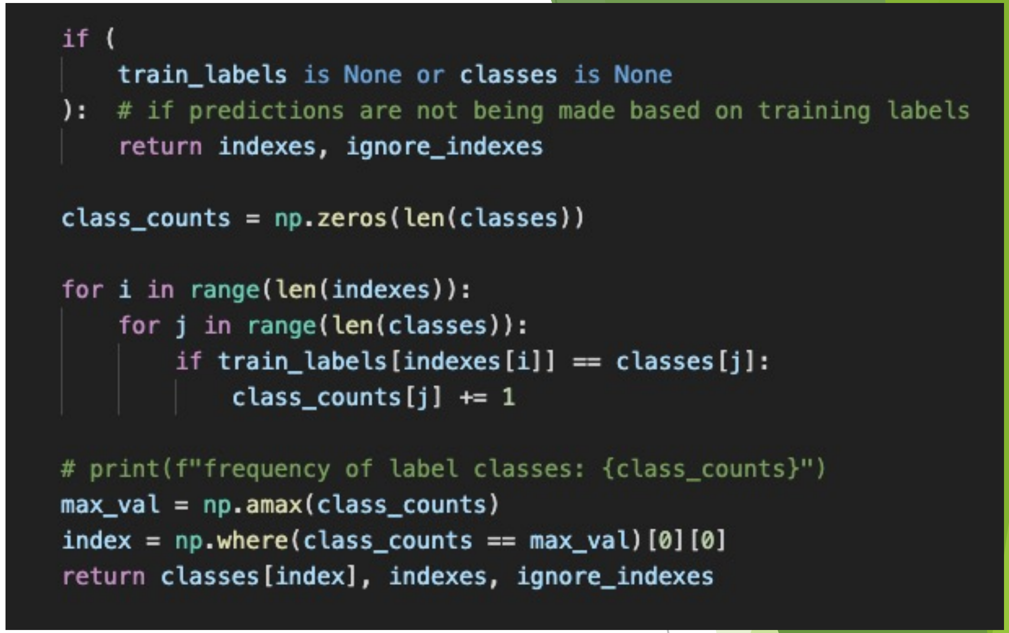
```python
def knn(train_data, test_data, k, train_labels=None, classes=None, missing_value=None):
    """
    knn algo
    """
    ignore_indexes = []
    if missing_value is not None:
        for i in range(len(test_data)):
            if test_data[i] == missing_value:
                ignore_indexes.append(i)

    diffs = []
    for i in range(len(train_data)):
        diff = []
        for j in range(len(train_data[i])):
            if j in ignore_indexes:
                continue
            diff.append(abs((train_data[i][j] - test_data[j])))
        diffs.append(diff)

    distances = []
    for i in range(len(diffs)):
        sum = 0
        for j in range(len(diffs[0])):
            sum += diffs[i][j] * diffs[i][j]
        distances.append(math.sqrt(sum))
    # print(distances)

    indexes = []
    for j in range(k):
        index = -1
        min = 100000000
        for i in range(len(distances)):
            if i in indexes:
                continue
            if distances[i] < min:
                min = distances[i]
                index = i
        indexes.append(index)

    # print(f"closest rows in train data: {indexes}")
```



```python
    if (
        train_labels is None or classes is None
    ):  # if predictions are not being made based on training labels
        return indexes, ignore_indexes

    class_counts = np.zeros(len(classes))

    for i in range(len(indexes)):
        for j in range(len(classes)):
            if train_labels[indexes[i]] == classes[j]:
                class_counts[j] += 1

    # print(f"frequency of label classes: {class_counts}")
    max_val = np.amax(class_counts)
    index = np.where(class_counts == max_val)[0][0]
    return classes[index], indexes, ignore_indexes
```

Classification Sets - Missing Values

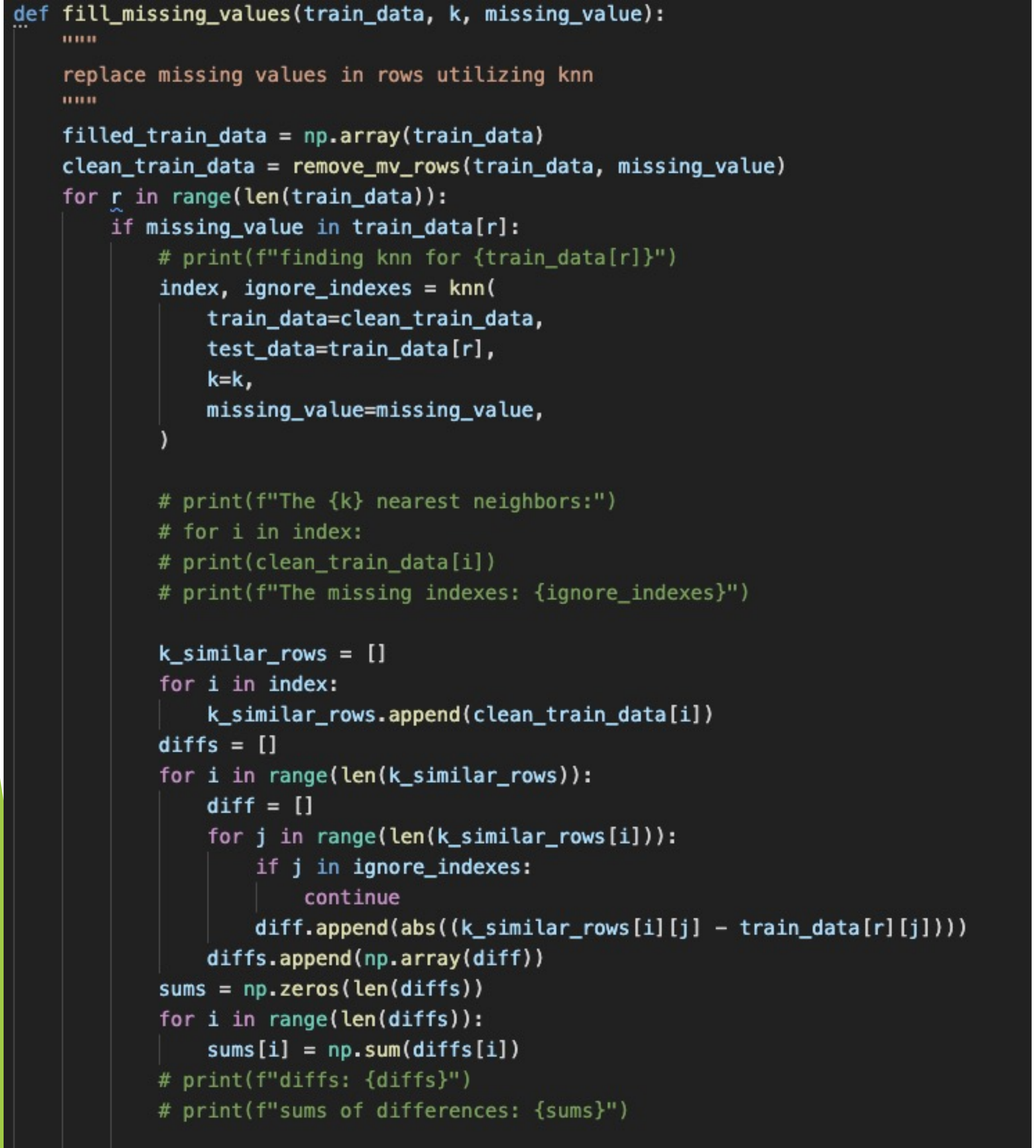
- TrainData 1 and TestData 1 contained missing values in every single row, so missing values were handled by calculating the mean of every column, and then fill the missing values in that column with the mean value.
- TrainData 3 and TestData 3 contained missing values, so missing values were handled by using KNN imputation (see part 2).

Part 2: Missing Value Estimation

- Missing values were calculated through KNN imputation.
- A function called *fill_missing_values* is defined, which takes in input data, the k parameter, and the missing value to be replaced.
- Firstly, a new array is created where all the rows are rows without missing values.
- This will be used as the training set for the KNN algorithm.
- Then, for every row r that contains missing values, the indexes of the k nearest neighbors are found.

Part 2: Missing Value Estimation

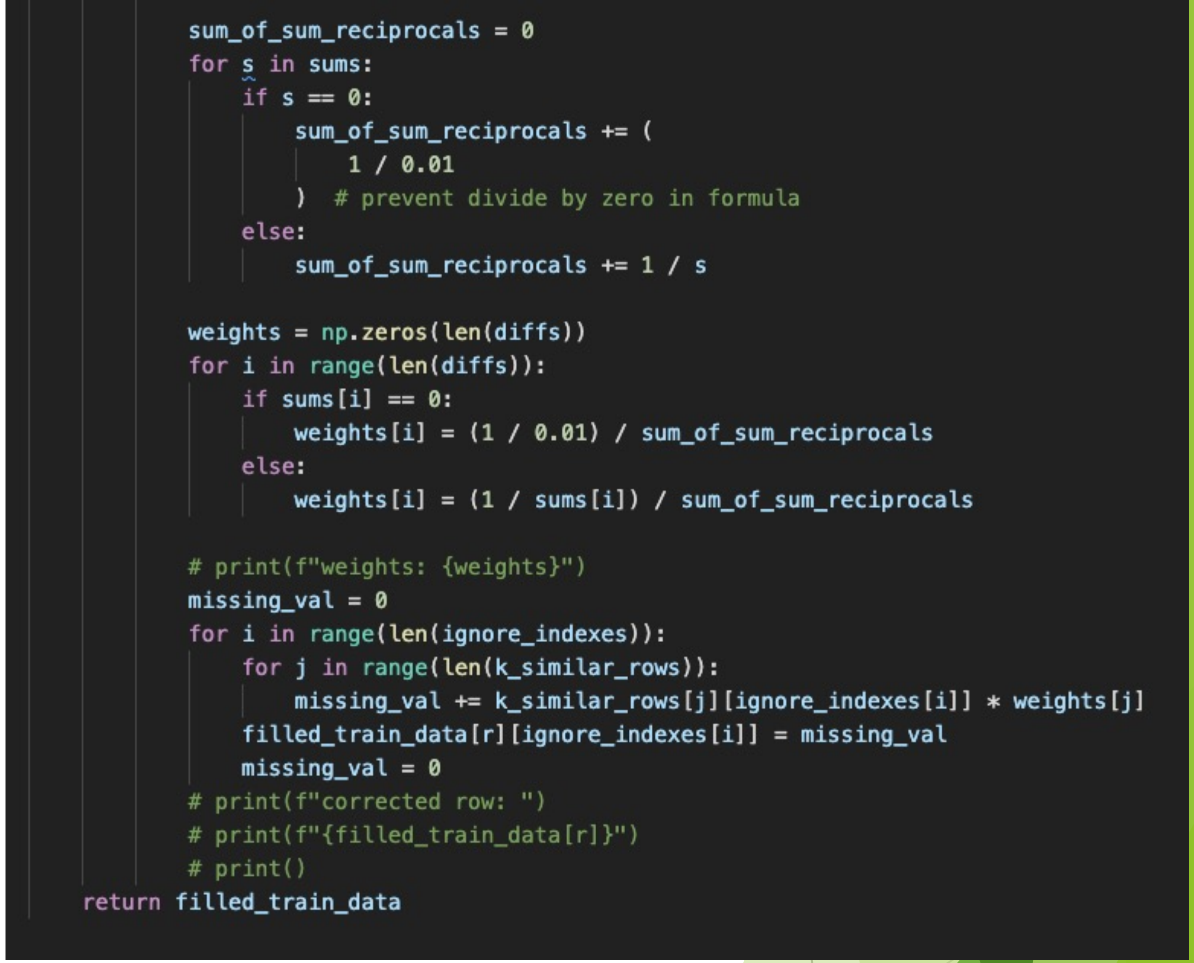
- For each missing attribute in the row, an estimate is created through a weighted sum which is calculated by summing the product of a neighbor i's non-missing attribute by a weight i, where the weight is calculated as follows:
- Let di equal the sum of the differences between each corresponding attribute in neighbor i and row r that does not belong to the column with the missing value.
- wi = $\frac{\frac{1}{d_i}}{\sum_j \frac{1}{dj}}$,
- where for every neighbor j to row r, dj is calculated in the same manner as di is calculated.
- Once every missing value is filled in the row, the same process will repeat for the next row with missing values, where the k nearest neighbors are found and estimates are calculated as shown above.
- This algorithm was applied to both MissingData1.txt and MissingData2.txt.



```python
def fill_missing_values(train_data, k, missing_value):
    """
    replace missing values in rows utilizing knn
    """
    filled_train_data = np.array(train_data)
    clean_train_data = remove_mv_rows(train_data, missing_value)
    for r in range(len(train_data)):
        if missing_value in train_data[r]:
            # print(f"finding knn for {train_data[r]}")
            index, ignore_indexes = knn(
                train_data=clean_train_data,
                test_data=train_data[r],
                k=k,
                missing_value=missing_value,
            )

            # print(f"The {k} nearest neighbors:")
            # for i in index:
            # print(clean_train_data[i])
            # print(f"The missing indexes: {ignore_indexes}")

            k_similar_rows = []
            for i in index:
                k_similar_rows.append(clean_train_data[i])
            diffs = []
            for i in range(len(k_similar_rows)):
                diff = []
                for j in range(len(k_similar_rows[i])):
                    if j in ignore_indexes:
                        continue
                    diff.append(abs((k_similar_rows[i][j] - train_data[r][j])))
                diffs.append(np.array(diff))
            sums = np.zeros(len(diffs))
            for i in range(len(diffs)):
                sums[i] = np.sum(diffs[i])
            # print(f"diffs: {diffs}")
            # print(f"sums of differences: {sums}")
```



```python
            sum_of_sum_reciprocals = 0
            for s in sums:
                if s == 0:
                    sum_of_sum_reciprocals += (
                        1 / 0.01
                    )  # prevent divide by zero in formula
                else:
                    sum_of_sum_reciprocals += 1 / s

            weights = np.zeros(len(diffs))
            for i in range(len(diffs)):
                if sums[i] == 0:
                    weights[i] = (1 / 0.01) / sum_of_sum_reciprocals
                else:
                    weights[i] = (1 / sums[i]) / sum_of_sum_reciprocals

            # print(f"weights: {weights}")
            missing_val = 0
            for i in range(len(ignore_indexes)):
                for j in range(len(k_similar_rows)):
                    missing_val += k_similar_rows[j][ignore_indexes[i]] * weights[j]
                filled_train_data[r][ignore_indexes[i]] = missing_val
                missing_val = 0
            # print(f"corrected row: ")
            # print(f"{filled_train_data[r]}")
            # print()
    return filled_train_data
```