CS 334: Homework #1 Solutions

1. **Numerical Programming**: The full code can be viewed in `q1.py`

   (a) Generate random numbers:

```python
def gen_random_samples():
    """
    Generate 5 million random samples using the
    numpy random.randn module.

    Returns
    ----------
    sample : 1d array of size 5 million
        An array of 5 million random samples
    """
    ## TODO FILL IN
    return np.random.randn(5000000)
```

   (b) Sum of squares using for-loop

```python
def sum_squares_for(samples):
    """
    Compute the sum of squares using a forloop

    Parameters
    ----------
    samples : 1d-array with shape n
        An array of numbers.

    Returns
    -------
    ss : float
        The sum of squares of the samples
    timeElapse: float
        The time it took to calculate the sum of squares (in seconds)
    """
    timeElapse = 0
    ss = 0
    startTime = time.time()
    for i in range(len(samples)):
        ss = ss + samples[i]*samples[i]
    timeElapse = time.time() - startTime
    return ss, timeElapse
```

   (c) Sum of squares using numpy

```python
def sum_squares_np(samples):
    """
    Compute the sum of squares using Numpy's dot module

    Parameters
    ----------
    samples : 1d-array with shape n
        An array of numbers.

    Returns
    -------
    ss : float
        The sum of squares of the samples
    timeElapse: float
        The time it took to calculate the sum of squares (in seconds)
    """
    timeElapse = 0
    ss = 0
```

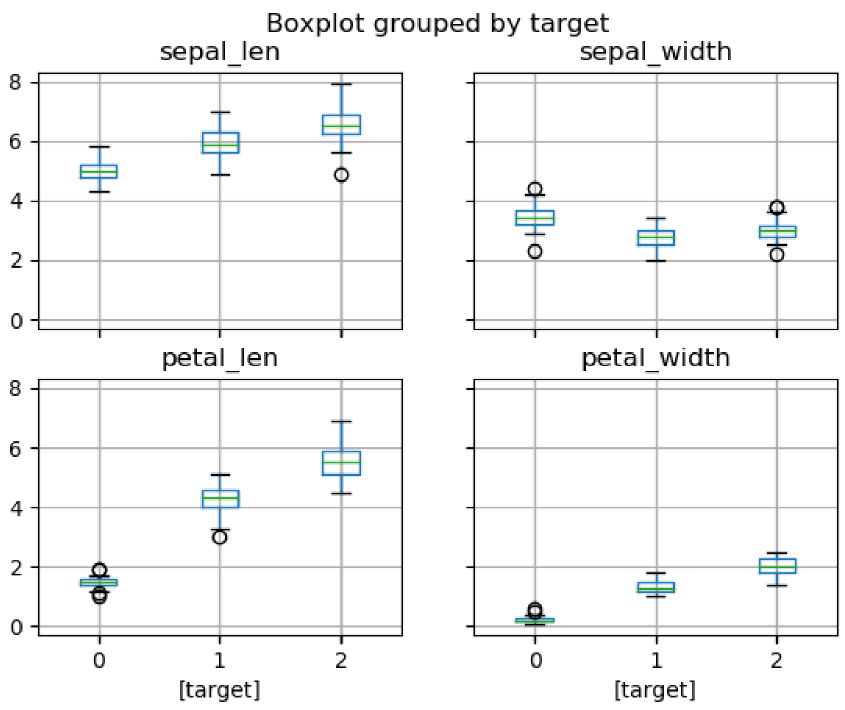


Figure 1: Pandas boxplot

```
    startTime = time.time()
    ss = np.dot(samples, samples)
    timeElapse = time.time() - startTime
    return ss, timeElapse
```

(d) Running `q1.py` from the command line yields:

```
>> python q1.py
Time [sec] (for loop): 1.7773489952087402
Time [sec] (np loop): 0.15200376510620117
```

Thus, the numpy loop is 10 times faster than the for loop.

2. **Visualization Exploration**: The full code can be viewed in `q2.py`

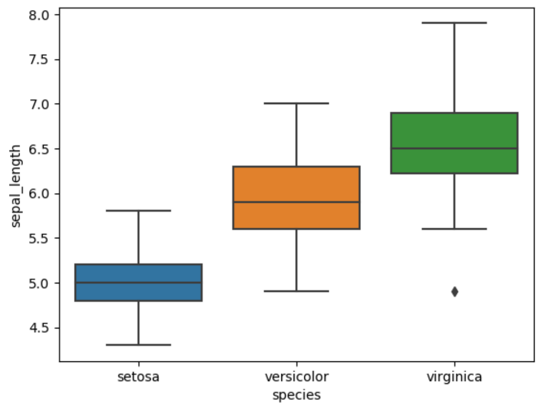
(a) Load the iris dataset

```
# load the iris dataset using sklearn
iris = datasets.load_iris()
```

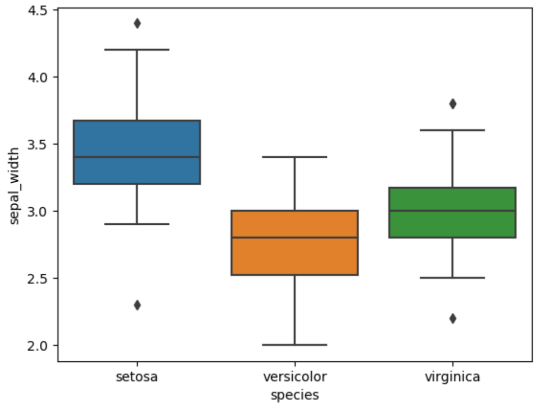
(b) Perform boxplot using Pandas (see Figure 1) or Seaborn (see Figure 2).

```
# pandas way to do boxplot the different ones and group by target
bpFig = irisDF.boxplot(column=['sepal_len', 'sepal_width',
                       'petal_len', 'petal_width'],
                       by=['target'])
plt.savefig('q2b-pd.png')
```

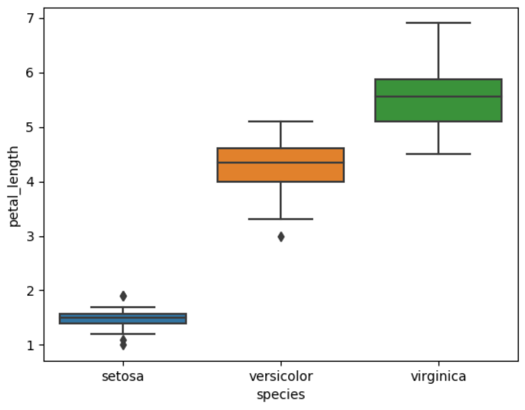
(c) Perform scatter plot using Pandas (see Figure 4) or Seaborn (see Figure 4).



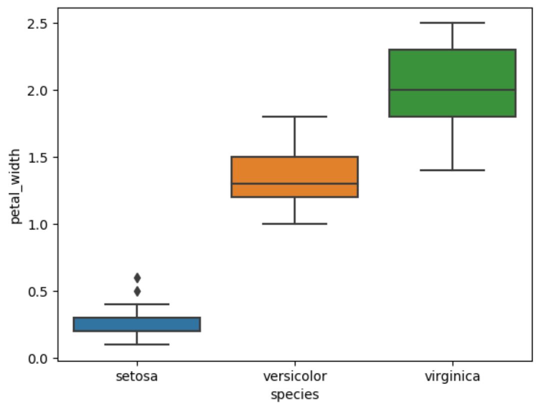
(a) Sepal Length



(b) Sepal Width



(c) Petal Length



(d) Petal Width

Figure 2: Boxplots using seaborn

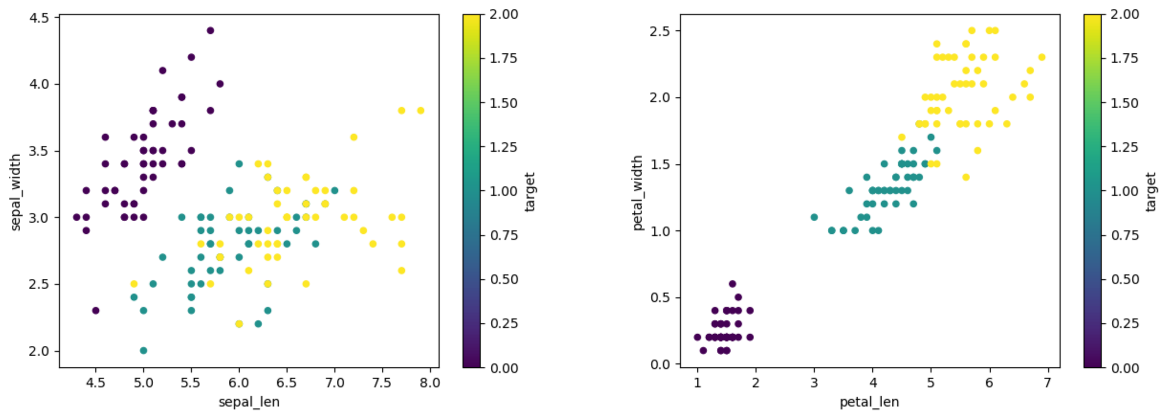


Figure 3: Scatterplot using Pandas

```python
# pandas way to do scatter plot
sepalPlt = irisDF.plot.scatter(x='sepal_len',
                               y='sepal_width',
                               c='target',
                               colormap='viridis')
plt.savefig('q2c-sepal-pd.png')
sepalPlt = irisDF.plot.scatter(x='petal_len',
                               y='petal_width',
                               c='target',
                               colormap='viridis')
plt.savefig('q2c-petal-pd.png')
```

3. **K-NN Implementation**

(a) For the `train` function, the idea was to store the features and labels as the training dataset. This meant adding at least one variable to the class so that you can reference it during the `predict` function.

```python
    def train(self, xFeat, y):
        """
        Train the k-nn model.

        Parameters
        ----------
        xFeat : nd-array with shape n x d
            Training data
        y : 1d array with shape n
            Array of labels associated with training data.

        Returns
        -------
        self : object
        """
        # store the two objects
        if isinstance(xFeat, pd.DataFrame):
            xFeat = xFeat.to_numpy()
        self.xFeat = xFeat
        self.y = y
        return self
```

(b) For the `predict` function, you needed to loop through each sample and then compare it against all the training dataset by calculating the distance to each point. One thing to

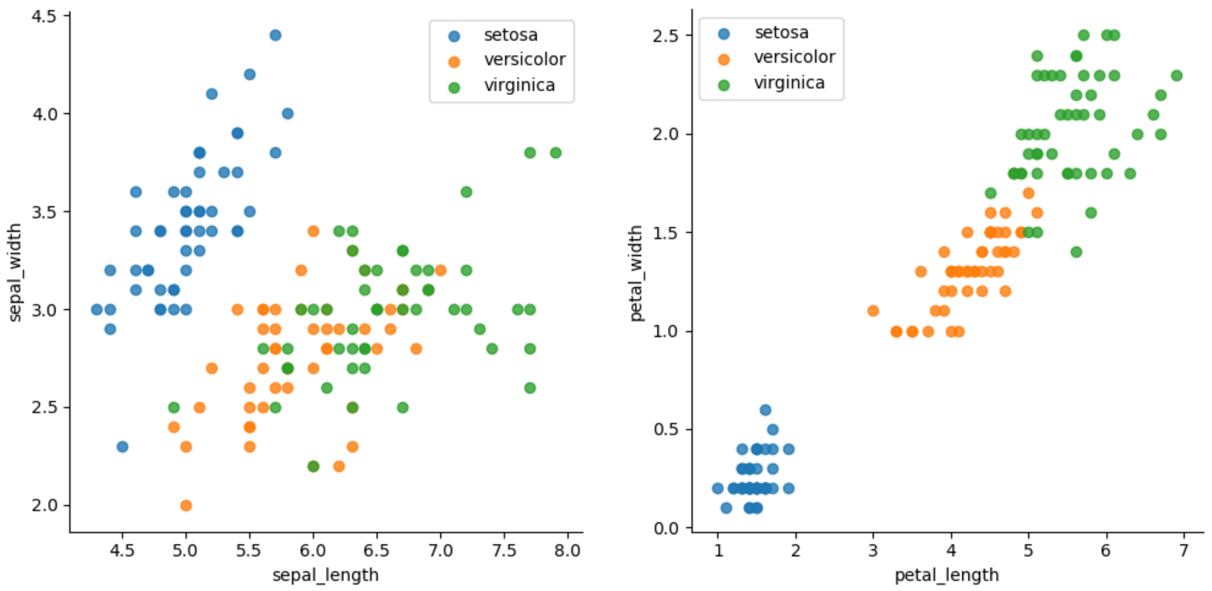


Figure 4: Scatterplot using Seaborn

note is that the Euclidean distance is equivalent to the ℓ_2-norm of the difference of the two samples $\mathbf{x}, \mathbf{z}$, which allows you to calculate it a bit more quickly than two for-loops. Note that to use `numpy` directly, we needed to cast it into an array to make it compatible, which is why there's a check for the datatype.

```
    def predict(self, xFeat):
        """
        Given the feature set xFeat, predict
        what class the values will have.

        Parameters
        ----------
        xFeat : nd-array with shape m x d
            The data to predict.

        Returns
        -------
        yHat : 1d array or list with shape m
            Predicted class label per sample
        """
        yHat = [] # variable to store the estimated class label
        # convert to numpy for ease
        if isinstance(xFeat, pd.DataFrame):
            xFeat = xFeat.to_numpy()
        # for each sample of the row
        for i in range(xFeat.shape[0]):
            # apply the euclidean distance which is just the 2-norm
            dist = np.linalg.norm(self.xFeat - xFeat[i, :], axis=1)
            # an equivalent way to do this would be:
            # tmp = (self.xFeat - xFeat[i, :])**2
            # dist = np.sqrt(np.sum(tmp, axis=1))
            # do an argument sort
            idx = np.argsort(dist)
            # get the labels for the first k
            yNeighbors = self.y.iloc[idx[0:self.k]]
            yHat.append(yNeighbors.mode()[0])
        return yHat
```

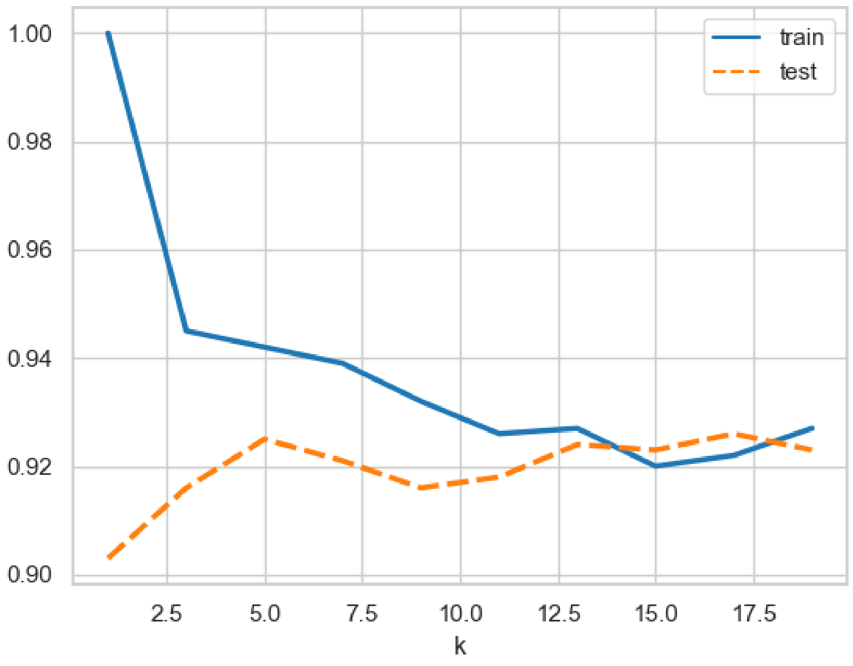


Figure 5: The accuracy on the y-axis as a function of k for k-nn.

(c) Accuracy can be computed by counting the number of times they vectors have the same value.

```python
def accuracy(yHat, yTrue):
    """
    Calculate the accuracy of the prediction

    Parameters
    ----------
    yHat : 1d-array with shape n
        Predicted class label for n samples
    yTrue : 1d-array with shape n
        True labels associated with the n samples

    Returns
    -------
    acc : float between [0,1]
        The accuracy of the model
    """
    # TODO calculate the accuracy
    acc = np.sum(yHat == yTrue) / len(yTrue)
    return acc
```

(d) We created a python script (`q3d.py`) that uses Seaborn to plot the accuracies as a function of k, which is shown in Figure 5. We tested only odd-values of k since this is a binary classification. Based on the plot, it seems that $k = 17$ might be a good value.

(e) The complexity of the predict function will vary based on how you implemented it, but for each sample of the test data, we need to compute the Euclidean distance, which is $O(nd)$ computations. Then our algorithm does a sort of n using quicksort $O(n \log n)$, so it's $O(nd + n \log n)$. Another common implementation will be $O(nd + kn)$ if you loop through all the training observations to figure out the top k.

4. **K-NN Preprocessing and performance**

   (a) Standard scale:

```
def standard_scale(xTrain, xTest):
    """
    Preprocess the training data to have zero mean and unit variance.
    The same transformation should be used on the test data. For example,
    if the mean and std deviation of feature 1 is 2 and 1.5, then each
    value of feature 1 in the test set is standardized using (x-2)/1.5.

    Parameters
    ----------
    xTrain : nd-array with shape n x d
        Training data
    xTest : nd-array with shape m x d
        Test data

    Returns
    -------
    xTrain : nd-array with shape n x d
        Transformed training data with mean 0 and unit variance
    xTest : nd-array with shape m x d
        Transformed test data using same process as training.
    """
    scaler = preprocessing.StandardScaler()
    scaler.fit(xTrain)
    return scaler.transform(xTrain), scaler.transform(xTest)
```

   (b) Minimax

```
def minmax_range(xTrain, xTest):
    """
    Preprocess the data to have minimum value of 0 and maximum
    value of 1.T he same transformation should be used on the test data.
    For example, if the minimum and maximum of feature 1 is 0.5 and 2, then
    then feature 1 of test data is calculated as:
    (1 / (2 - 0.5)) * x - 0.5 * (1 / (2 - 0.5))

    Parameters
    ----------
    xTrain : nd-array with shape n x d
        Training data
    xTest : nd-array with shape m x d
        Test data

    Returns
    -------
    xTrain : nd-array with shape n x d
        Transformed training data with min 0 and max 1.
    xTest : nd-array with shape m x d
        Transformed test data using same process as training.
    """
    scaler = preprocessing.MinMaxScaler()
    scaler.fit(xTrain)
    return scaler.transform(xTrain), scaler.transform(xTest)
```

   (c) Adding irrelevant features

```
def add_irr_feature(xTrain, xTest):
    """
```

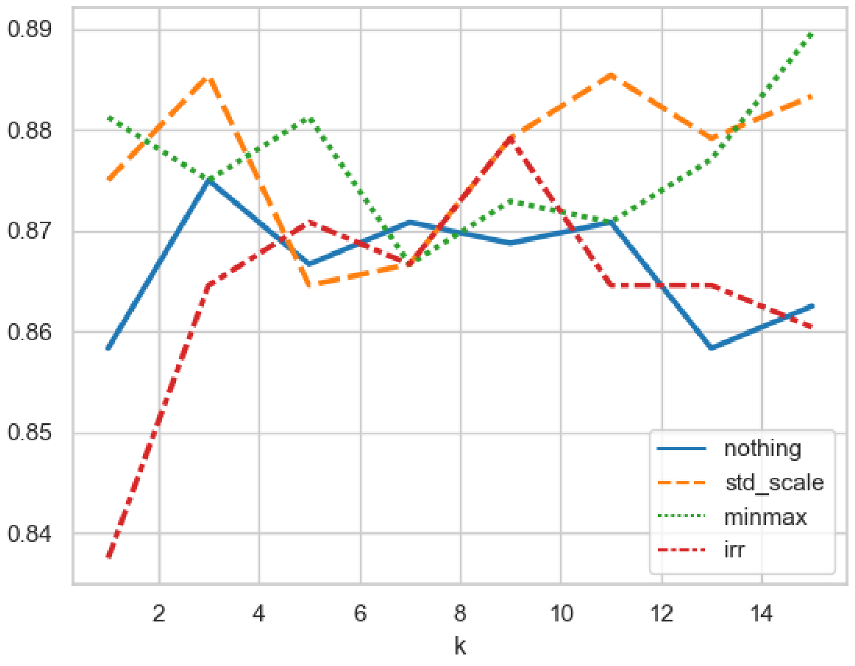


Figure 6: The accuracy on the y-axis as a function of k for k-nn.

```
Add 2 features using Gaussian distribution with 0 mean,
standard deviation of 1.

Parameters
----------
xTrain : nd-array with shape n x d
    Training data
xTest : nd-array with shape m x d
    Test data

Returns
-------
xTrain : nd-array with shape n x (d+2)
    Training data with 2 new noisy Gaussian features
xTest : nd-array with shape m x (d+2)
    Test data with 2 new noisy Gaussian features
"""
xTrCopy = xTrain.copy()
xTrCopy['irr1'] = np.random.normal(scale=1, size=xTrain.shape[0])
xTrCopy['irr2'] = np.random.normal(scale=1, size=xTrain.shape[0])
xTestCopy = xTest.copy()
xTestCopy['irr1'] = np.random.normal(scale=1, size=xTest.shape[0])
xTestCopy['irr2'] = np.random.normal(scale=1, size=xTest.shape[0])
return xTrCopy, xTestCopy
```

(d) We created a python script that uses Seaborn to plot the accuracies as a function of k, which is shown in Figure 6. Generally speaking, adding irrelevant features seems to hurt the accuracy for smaller values of K, while pre-processing the data using standard scale and min+max range overall seems to help.