Homework #3 A

Spring 2020, CSE 446/546: Machine Learning
Dino Bektesevic

Collaborated: Conor Sayers, Joachim Moeyenes, Jessica Birky, Leah Fulmer

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few word

a. *[2 points]* True or False: Given a data matrix $X \in \mathbb{R}^{n \times d}$ where $d$is much smaller than n, if we project our data onto a k dimensional subspace using PCA where $k = \text{rank}X$, our projection will have 0 reconstruction error (we find a perfect representation of our data, with no information loss)
True. The rank of X is the number of non-zero eigenvalues, i.e. the maximal number of linearly independent rows. By definition this is maximally d, since $d << n$. The remaining entries in X can then be represented as a linear combination of these vectors.

b. *[2 points]* True or False: The maximum margin decision boundaries that support vector machines construct have the lowest generalization error among all linear classifiers.
False.

c. *[2 points]* True or False: An individual observation x_i can occur multiple times in a single bootstrap sample from a dataset X, even if x_i only occurs once in X.
True. Bootstrap with replacements does exactly this, it leaves the picked data in the dataset from which we pick again, so it's quite possible to sample a pont twice.

d. *[2 points]* True or False: Suppose that the SVD of a square $n \times n$ matrix X is USV^T, where S is a diagonal $n \times n$ matrix. Then the rows of V are equal to the eigenvectors of $X^T X$.
False. Columns are eigenvectors, not rows. This follows from the definition of V in SVD.

e. *[2 points]* True or False: Performing PCA to reduce the feature dimensionality and then applying the Lasso results in an interpretable linear model.
Not necessarily. For example, in astronomy, doing PCA on galaxy spectra usually recovers the spectral continuum as the zeroth PCA components. Following component are spiky and tend to isolate spectral emission lines, usually where H emission is, and sometimes some components can even capture a known source of noise in the data. I guess then a linear combination of the recovered features would give you an interpretable model (e.g. zeroing out the noise while adding the remaining ones recovers a "clean" spectra, i.e. physically interpretable spectra). This is, however, not the case in general. In the case of spectra it's because the total galaxy spectra is a just a linear superposition of all the sources. If that weren't linear I suspect this scheme wouldn't work that well.

f. *[2 points]* True or False: choosing k to minimize the k-means objective (see Equation (1) below) is a good way to find meaningful clusters.
False, unless we are familiar with the system we are modeling there is no reason to believe k-means will recover meaningful clusters.

g. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u, v) = \exp(\frac{-||u-v||_2^2}{2\sigma^2})$). It seems to under-fit the training set: should you increase or decrease σ?

Kernels and the Bootstrap

A2. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose i-th component is

$$\phi_i(x) = \frac{1}{\sqrt{i!}} e^{\frac{-x^2}{2}} x^i$$

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.) Show that $K(x, x') = e^{-1/2(x-x')^2}$ is a kernel function for this feature map, i.e., $\phi(x) \cdot \phi(x') = e^{-1/2(x-x')^2}$.
Hint:Use the Taylor expansion of e. (This is the one dimensional version of the Gaussian (RBF) kernel).

$$\begin{aligned}
\phi_i(x) \cdot \phi_i(x') &= \left[\frac{1}{\sqrt{i!}} e^{\frac{-x^2}{2}} x^i\right] \cdot \left[\frac{1}{\sqrt{i!}} e^{\frac{-x'^2}{2}} x'^i\right] \\
&= \sum_i^\infty \frac{1}{i!} x^i x'^i e^{\frac{-x^2-x'^2}{2}} \\
&= e^{\frac{-x^2-x'^2}{2}} \sum_i^\infty \frac{x^i x'^i}{i!} \\
&= e^{\frac{-x^2-x'^2}{2}} e^{-xx'} \\
&= e^{\frac{-x^2}{2} - \frac{x'^2}{2} - xx'} \\
&= e^{-\frac{(x-x')^2}{2}}
\end{aligned}$$

Where we recognized that the sum resulting from writing out the dot product is the Taylor expansion of exponential function.

A3. This problem will get you familiar with kernel ridge regression using the polynomial and RBF kernels. First, let's generate some data. Let $n = 30$ and $f_*(x) = 4\sin(\pi x)\cos(6\pi x^2)$. For $i = 1, \ldots, n$ let each x_i be drawn uniformly at random on $[0, 1]$ and $y_i = f_*(x_i) + i$ where $i \approx \mathcal{N}(0, 1)$. For any function f, the true error and the train error are respectively defined as

$$\varepsilon_{\text{true}}(f) = E_{XY}[(f(X) - Y)^2]$$

$$\varepsilon_{\text{train}}(f) = \frac{1}{n}\sum_{i=1}^{n} = (f(x_i) - y_i)$$

Using kernel ridge regression, construct a predictor

$$\alpha = \underset{\alpha}{\text{argmin}}\, ||K\alpha - y||^2 + \lambda\alpha^T K\alpha$$

$$\hat{f}(x) = n\sum_{i=1}^{k} \widehat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Include any code you use for your experiments in your submission.

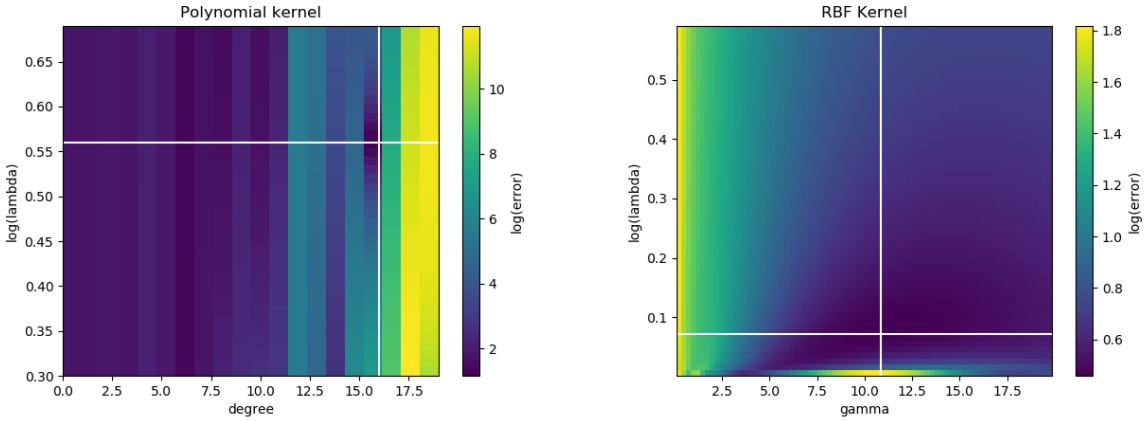
a. *[10 points]* Using leave-one-out cross validation, find a good $\lambda$and hyperparameter settings for the following kernels:

   (a) $k_{poly}(x, z) = (1 + x^T z)^d$ where $d \in \mathcal{N}$ is a hyperparameter,

   (b) $k_{rbf}(x, z) = \exp(-\gamma||x - z||^2$ where $\gamma > 0$ is a hyperparameter,

   Report the values of d,γ, and the λ values for both kernels.

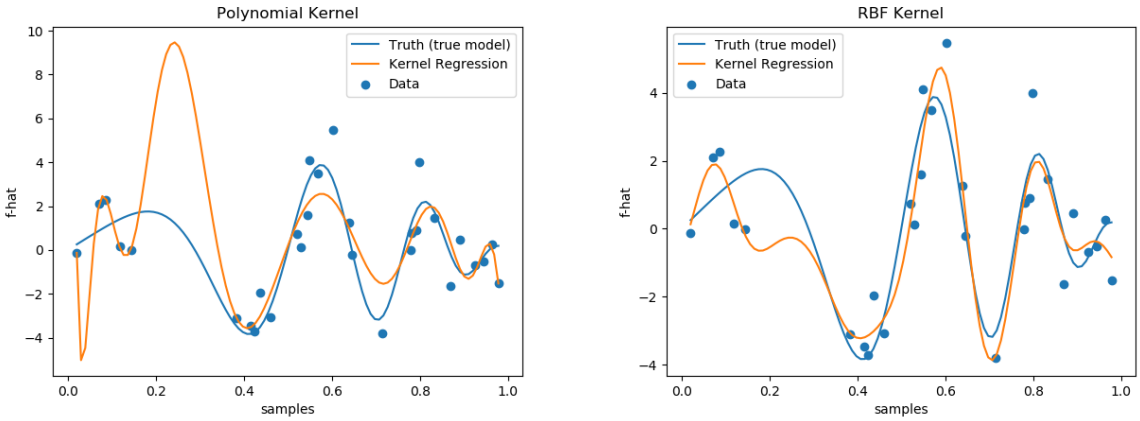
   For polynomial kernel I find optimal lambda: 0.56, degree: 16.0 sampled @ minimal error: 3.165 (log(Err)=1.152). For RBF kernel I find optimal 0.071, gamma: 10.85 sampled @ minimal error: 1.58 (log(Err)=0.4577).



(a) (b)

Figure 1: Cross validated error sampled at different pairs of regularization (y-axis) and hyper (x-axis) parameters. White cross marks the minimal sampled cross validation error.

b. *[10 points]* Let $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ be the functions learned using the hyperparameters you found in part a. For a single plot per function $\widehat{f} \in \{\widehat{f}_{\text{poly}}(x), \widehat{f}_{\text{poly}}(x)\}$, plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$,and $\widehat{f}(x)$ (i.e., define a fine grid on $[0, 1]$ to plot the functions).

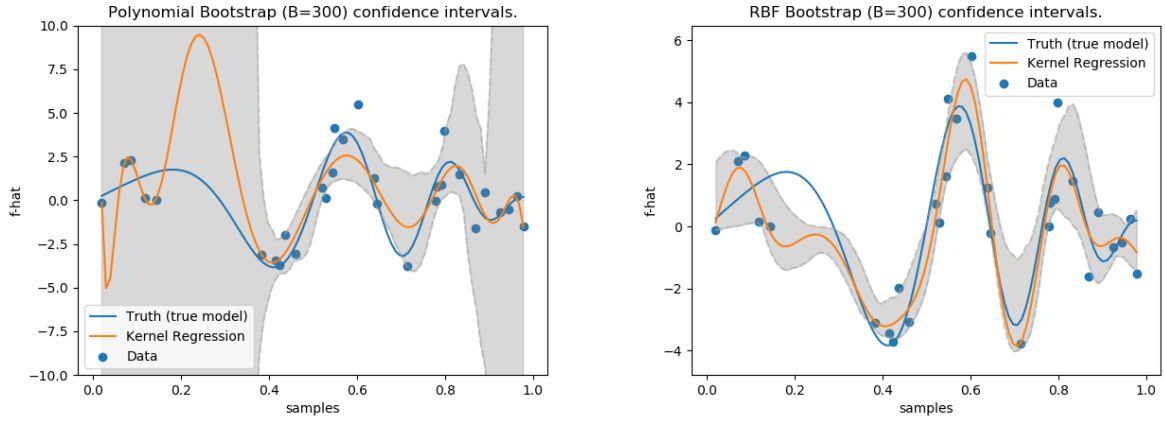


(a) (b)

Figure 2: Kernels evaluated on [0,1] using the best fit values of regularization and hyper parameters.

c. *[5 points]* We wish to build bootstrap percentile confidence intervals for $\widehat{f}_{\text{poly}}(x)$ and $\widehat{f}_{\text{rbf}}(x)$ for all $x \in [0, 1]$ from part b. Use the non-parametric bootstrap with $B = 300$ bootstrap iterations to find 5% and 95% percentiles at each point x on a fine grid over $[0, 1]$.

Specifically, for each bootstrap sample $b \in \{1, ..., B\}$, draw uniformly at randomly with replacement n samples from $\{(x_i, y_i)\}_{i=1}^n$, train an $\widehat{f}_n$ using the b-th resampled dataset, compute $\widehat{f}_b(x)$ for each x in your fine grid; let the 5th percentile at point x be the largest value ν such that $B\sum_{b=1}^B \mathbf{1}\{\widehat{f}_b(x) \leq \nu\} leq .05$, define the 95% analogously. Plot the 5 and 95 percentile curves on the plots from part b.

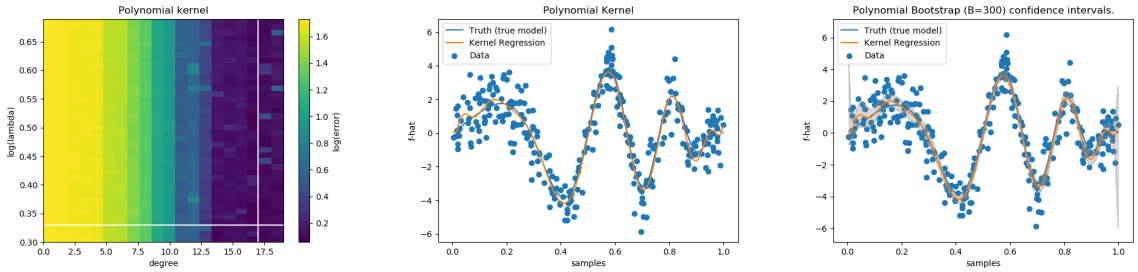


(a) (b)

Figure 3: Bootstrapped 5th and 95th percentile confidence interval.

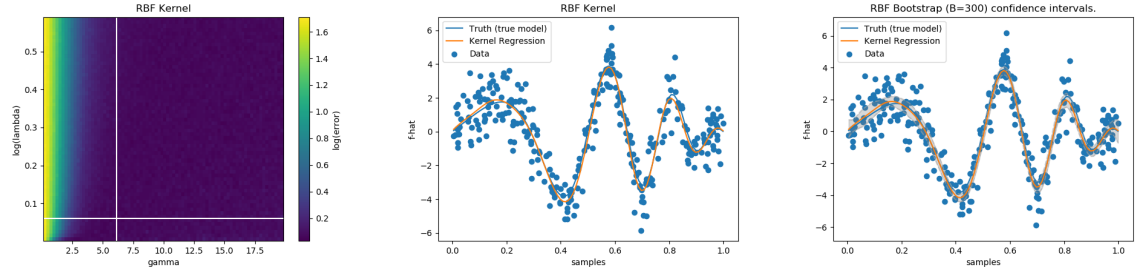
d. *[5 points]* Repeat parts a, b, and c with $n = 300$, but use 10-fold CV instead of leave-one-out for part a.

For polynomial kernel I find optimal lambda: 0.33, degree: 17.0 sampled @ minimal error: 1.059 (log(Err)=0.05697). For RBF kernel I find optimal lambda: 0.061, gamma: 6.1 sampled @ minimal error: 1.03 (log(Err)=0.02964). These numbers indicate that the RBF fits in a) are not the best because there are significant changes between the fitted parameters, but the ones for the polynomial fit seem to be stable.



(a) Cross validation error (b) Best fitting kernel plot. (c) Bootstrapped CIs

Figure 4: Part a-c using Polynomial kernel



(a) Cross validation error (b) Best fitting kernel plot. (c) Bootstrapped CIs

Figure 5: Part a-c using RBF kernel

e. *[5 points]* For this problem, use the $\widehat{f}_{\text{poly}}$ and $\widehat{f}_{\text{rbf}}$ learned in part d. Suppose $m = 1000$ additional samples $(x'_1, y'_1), \ldots, (x'_m, y'_m)$ are drawn i.i.d. the same way the first n samples were drawn. Use the non-parametric bootstrap with $B = 300$ to construct a confidence interval on $E[(Y - \widehat{f}_{\text{poly}}(X))^2 - (Y - \widehat{f}_{\text{rbf}})^2]$ (i.e. randomly draw with replacement m samples denoted as $\{(\tilde{x}'_i, \tilde{y}'_i)\}_{i=1}^m$ from $\{(\tilde{x}'_i, \tilde{y}'_i)\}_{i=1}^m$ and compute

$$\frac{1}{m}\sum_{i=1}^{m}\left[(\tilde{y}'_i - \widehat{f}_{\text{poly}}(\tilde{x}'_i))^2 - (\tilde{y}'_i - \widehat{f}_{\text{rbf}}(\tilde{x}'_i))^2\right]$$

repeat this B times) and find 5% and 95% percentiles. Report these values.Using this confidence interval, is there statistically significant evidence to suggest that one of $\widehat{f}_{\text{poly}}$ $\widehat{f}_{\text{rbf}}$ is better than the other at predicting Y from X? (Hint: does the confidence interval contain 0?)

I retrieve the 5th and 95th percentile interval of the given expression to be from 0.001955 to 0.023875. The interval does not contain zero, therefore we conclude we are 95% confident that there is a statistically significant evidence in favor of fitting with the RBF kernel.

The above plots and numbers were produced by the attached code. Note that the values of searched parameters were retrieved by manually scanning smaller and smaller intervals, using a finer and finer grid. This is generally considered bad (there are no guarantees that there are no other local minima accidentally "skipped" over during gridding). I froze the random seed at 0 so that I could at least compare the results with my colleagues.

```
import numpy as np
import matplotlib.pyplot as plt
import itertools
import multiprocessing as mp
import time

np.random.seed(0)

class Kernel:
    """Base class used to define a generic, actionless, kernel (KO) to use in
    ridge regression defined as:

        a = argmin ||K*a - y||^2 + l*a^T*K*a
    where a is alpha, l - lambda and K the kernel itself and K=k(x_i, x_j) an
    kernel action. Kernel action is defined by its eval method, which must be
    overriden by subclass.

    Attributes
    ----------
    x: `np.array`, optional
        Feature space.
    y: `np.array`, optional
        Labels.
    lambd: `float`, optional
        Regularization parameter
    hyperparams: `dict`, optional
        A dictionary of hyperparameters.
    alpha: `dict`
        Lerned predictor (learned weights).

    Notes
    -----
    Fitting/training the kernel will update, replace, its features and labels
    with the ones given for training.

    Hyperparameters must be registered in the hyperparameters dictionary when
    inheriting.
    """
    def __init__(self, x=None, y=None, lambd=None, **kwargs):
        self.update(x=x, y=y, lambd=lambd, **kwargs)
        self.alpha = None

    def update(self, **kwargs):
        """Update given class attributes (x, y, lambd, hyperparams)."""
        x = kwargs.pop("x", None)
        y = kwargs.pop("y", None)
        lambd = kwargs.pop("lambd", None)
        hyperparams = kwargs

        self._mean = None
        self._std = None

        if x is not None:
            self.x = x
        if y is not None:
            self.y = y
        if lambd is not None:
            self.lambd = lambd
        if len(hyperparams) != 0:
            self.hyperparams = hyperparams

    def eval(self, *args, **kwargs):
        """Defines kernel action, i.e. given an x_i, x_j evaluates the kernel.
        Needs to be an vectorized operation, such that supplying two vectors
        with dimensions n and m returns n-by-m matrix.
        """
        raise NotImplementedError

    def standardize(self, x, mean=None, std=None):
        """Returns a standardized copy of the array using the given weights.
        """
        if mean is None:
            self._mean = np.mean(x)
            mean = self._mean
        if std is None:
            self._std = np.std(x)
            std = self._std
        return (x-mean)/std

    def fit(self, x, y):
        """Given features x and labels y, updates the class attributes and
        preforms ridge regression. Stores the learned predictor in the alpha
        attribute.

        Parameters
        ----------
        x: `np.array`
            Features to train on.
        y: `np.array`
            Labels for corresponding features.
        """
        self.update(x=x, y=y)
        x = self.standardize(x)
        K = self.eval(x, x)
        self.alpha = np.linalg.solve(K + self.lambd * np.eye(len(K)), y)

    def predict(self, x):
        """Using learned weights and given features predicts the labels.

        Parameters
        ----------
        x: `np.array`
            Features.

        Returns
        --------
        y: `np.array`
            Predicted labels

        Raises
```

```
        ------
        AttributeError
            When kernel has not been trained and thus has no alpha attributed.
            Call fit with the features and associated labels to train the
            kernel.
        """
        if self.alpha is None:
            raise AttributeError("Kernel has not been trained.")
        sx = self.standardize(self.x, self._mean, self._std)
        xx = self.standardize(x, self._mean, self._std)
        kernel = self.eval(sx, xx)
        return np.dot(self.alpha, kernel)

    def score(self, x, y):
        """Predicts the labels of the given features and compares them to the
        given truth (true labels). Associates a score to the \"goodness\" of
        prediction.

        Parameters
        ----------
        x: `np.array`
            Features for which labels will be predicted.
        y: `np.array`
            Truth, true labels for the corresponding features.

        Returns
        -------
        score: `float`
            Mean of the square of differences in predicitons, the score of
            the goodness of predictions.
        """
        return np.mean((self.predict(x) - y)**2)


class PolynomialKernel(Kernel):
    """Class implementing the polynomial kernel."""

    def __init__(self, lambd, degree, **kwargs):
        """Instantiate a polynomial kernel.

        Parameters
        ----------
        lambd: `float`
            Regularization parameter
        degree: `int`
            Degree of the polynomial
        """
        super().__init__(lambd=lambd, degree=degree, **kwargs)

    @property
    def d(self):
        return self.hyperparams["degree"]

    def eval(self, x, z):
        """Evaluate the kernel on given points. Kenrel action is defined as

            K(x,z) = (1 + x*z)**d

        Parameters
        ----------
        x: `np.array`
            A column vector of features.
        z: `np.array`
            A column vector of ``n`` points at which to evaluate the kernel.

        Returns
        -------
        eval: `np.array`
            An n-by-d matrix where each column is the kernel evaluated at a
            single point.
        """
        return (1 + np.outer(x, z))**self.d


class RBFKernel(Kernel):
    """Class implementing the RBF kernel."""

    def __init__(self, lambd, gamma, **kwargs):
        """Instantiate an RBF kernel.

        Parameters
        ----------
        lambd: `float`
            Regularization parameter
        gamma: `int`
            Degree of the polynomial
        """
        super().__init__(lambd=lambd, gamma=gamma, **kwargs)

    @property
    def gamma(self):
        return self.hyperparams["gamma"]

    def eval(self, x, z):
        """Evaluate the kernel on given points. Kernel action is defined as

            K(x,z) = exp(-gamma*(x-z)^2)

        Parameters
        ----------
        x: `np.array`
            A column vector of features.
        z: `np.array`
            A column vector of ``n`` points at which to evaluate the kernel.

        Returns
        -------
        eval: `np.array`
            An n-by-d matrix where each column is the kernel evaluated at a
            single point.
```

```
        """
        return np.exp(-self.gamma * np.subtract.outer(x, z)**2)


class KernelFactory:
    """Kernel factory."""

    @staticmethod
    def create(kernelType, *args, **kwargs):
        """Given either ``poly`` or `rbf`` kernel types and arguments returns
        an instance of PolynomialKernel or RBFKernel.

        Parameters
        ----------
        kernelType: `str`
            A string containing either ``poly`` or ``rbf`` targeting which
            kernel to isntantiate
        args
        kwargs
            All other arguments are passed to the class instntiation call.
        """
        if "poly" in kernelType.lower():
            return PolynomialKernel(*args, **kwargs)
        elif "rbf" in kernelType.lower():
            return RBFKernel(*args, **kwargs)


def truth(x):
    """The truth, the true model that we are trying to reconstruct. A function

        f(x) = 4*sin(pi*x) * cos(6*pi*x^2)

    Parameters
    ----------
    x: `np.array`
        Array of points in which to evaluate the function

    Returns
    -------
    y: `np.array`
        Function values (i.e. labels in this context)
    """
    return 4 * np.sin(np.pi * x) * np.cos(6 * np.pi * x**2)


def cross_validate(kernel, x, y, foldSize):
    """Performs cross validation of the kernel on the given dataset by randomly
    permuting the order of features, training on the fold-sized subsets of the
    total dataset {x_i, y_i} and then scoring the predictions of the trained
    kernel.

    Parameters
    ----------
    kernel: `object`
        A Kernel subclass.
    x: `np.array`
        Features
    y: `np.array`
        Labels
    foldSize: `int`
        The size of the selected subsets of data to train on

    Returns
    -------
    meanScore: `float`
        The mean of all of the scores scored after training on random subsets.
    """
    n = len(x)
    idxs = np.random.permutation(np.arange(0, n))

    if foldSize == 0:
        kernel.fit(x, y)
        scores = np.array([kernel.score(x, y)])
    else:
        scores = np.zeros(foldSize)
        for i in range(foldSize):
            lower = int(n/foldSize * i)
            upper = int(n/foldSize * (i+1))

            xPredict = x[idxs[lower:upper]]
            yPredict = y[idxs[lower:upper]]
            xFit = np.concatenate([x[idxs[0:lower]], x[idxs[upper:]]])
            yFit = np.concatenate([y[idxs[0:lower]], y[idxs[upper:]]])

            kernel.fit(xFit, yFit)
            scores[i] = kernel.score(xPredict, yPredict)

    return np.mean(scores)


def sampler(x, y, lambdas, hyperparams, foldSize, kernelType):
    """Preforms grid sampling of cross evaluated scores over all pairs of
    given lambdas and hyperparameters.

    Parameters
    ----------
    x: `np.array`
        Features
    y: `np.array`
        Labels
    lambdas: `np.array`
        Regularization parameters at which to preform cross validation.
    hyperparams: `np.array`
        Hyperparameter values at which to preform cross validation.
    foldSize: `int`
        Cross validation folding size.
    kernelType: `str`
        Which kernel type to use.

    Returns
```

```
    -------
    samples: `np.array`
        A structured numpy array containing the cross validation error, log
        of regularization parameters and hyperparameter values at which the
        error was measured.
    bestFit: `dict`
        A dictionary containing the minimal sampled cross validation error and
        the values of lambda and hyperparameter at that error.
    """
    nCombinations = len(lambdas)*len(hyperparams)
    dt = [("lambda", float), ("hyperparams", float), ("error", float)]
    samples = np.zeros((nCombinations, ), dtype=dt)

    for i, (lambd, hparam) in enumerate(itertools.product(lambdas, hyperparams)):
        k = KernelFactory.create(kernelType, lambd, hparam)
        samples["error"][i] = cross_validate(k, x, y, foldSize)
        samples["lambda"][i] = lambd
        samples["hyperparams"][i] = hparam

    idxMinErr = np.where(samples['error'] == samples['error'].min())[0][0]
    minErr = samples['error'][idxMinErr]
    optimLambda = samples['lambda'][idxMinErr]
    optimHParam = samples['hyperparams'][idxMinErr]

    print(f"Using {kernelType} with fold size={foldSize}: ")
    print(f"    Optimal lambda: {optimLambda:.4}, hyperparam: {optimHParam:.4} "
          f"sampled @ minimal error: {minErr:.4} (log(Err)={np.log(minErr):.4}).")

    return samples, {"minErr": minErr, "lambda": optimLambda,
                     "hyperparam": optimHParam}


def plotA3a(fig, ax, x, y, z, bestFit, xlabel="Hyperparameter",
            ylabel="log(lambda)", cbarlbl="log(error)", title="Kernel"):
    """Plots the grid sampled cross validation errors as a function of sampled
    regularization and hyper-parameters.


    Parameters
    ----------
    fig: `matplotlib.pyplot.Figure`
       Figure to which a colorbar will be attached.
    ax: `matplotlib.pyplot.Axes`
       Axis on which to plot
    x: `np.array`
        Features, a vector of length n
    y: `np.array`
        Labels, a vector of length m
    z: `np.array`
        An m*n lenght array of cross-validation errors.
    bestFit: `dict`
        dictionary containing the minimal error, and the values of
        regularization and hyper-parameters at that point.
    xlabel: `str`, optional
        X axis label. Defaults to hyperparameter
    ylabel: `str`, opitonal
        Y axis label. Defaults to log(lambda)
    cbarlbl: `str`, optional
        Colorbar label. Defaults to log(error)
    title: `str`, optional
        Axis title. Defaults to "Kernel".

    Returns
    -------
    ax: `matplotlib.pyplot.Axes`
        Modified axis containing the plot.
    cbar: `matplotlib.pyplot.Axes`
        Axis containing the colorbar.
    """
    smpls = z.reshape(len(y), len(x))

    img = ax.imshow(smpls, interpolation=None, aspect="auto",
                    extent=(x[0], x[-1], y[0], y[-1]), origin="lower")
    cbar = fig.colorbar(img, ax=ax, orientation='vertical')
    ax.axvline(bestFit['hyperparam'], color="white")
    ax.axhline(bestFit['lambda'], color="white")

    cbar.set_label(cbarlbl)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)

    return ax, cbar


def A3a(kernelType, x, y, foldSize, lambdas, hyperparams, xlabel="", title=""):
    """Samples the cross validation error on a grid for both polynomial and RBF
    kernels. Reports the minimal found error values and plots the errors.

    Parameters
    ----------
    kernelType: `str`
        Kernel type (``poly`` or ``rbf``)
    x: `np.array`
        Features
    y: `np.array`
        Labels
    foldSize: `int`
        Cross validationsubset size
    lambdas: `np.array`
        Regularization parameters at which to calculate cross validation error.
    hyperparams: `np.array`
        Hyperparameters at which to calculate cross validation error.
    xlabel: `str`
        X axis label
    title: `str`
        Axis title
    """
    samples, best = sampler(x, y, lambdas, hyperparams, foldSize=foldSize,
                            kernelType=kernelType)
```

```
    fig, axes = plt.subplots()
    plotA3a(fig, axes, hyperparams, lambdas, np.log(samples['error']),
            best, xlabel=xlabel, title=title)

    return samples, best


def A3b(kernelType, x, y, bestFit, axis=None, title=None, xlabel="samples", ylabel="f-hat"):
    """Using best fit parameters plots the data, the truth (true model) and the
    best fitting kernel.

    Parameters
    ----------
    kernelType: `str`
        Kernel type to use (``poly`` or ``rbf``)
    x: `np.array`
        Features (train data set)
    y: `np.array`
        Labels (train data set)
    bestFit: `dict`
        A dictionary containing the minimal sampled cross validation error and
        the values of lambda and hyperparameter at that error.
    axis: `matplotlib.pyplot.Axes`
        Axis on which to plot, otherwise a new figure will be created.
    title: `str`
        Title to use.
    """
    kernel = KernelFactory.create(kernelType, bestFit['lambda'], bestFit['hyperparam'])
    kernel.fit(x, y)

    if title is None:
        title = ""
    if axis is None:
        fig, axis = plt.subplots()

    # we need more evenly spaced arrays for plots, otherwise ugly
    xTest = np.linspace(x.min(), x.max(), 100)
    yHat = kernel.predict(xTest)

    axis.scatter(x, y, label="Data")
    axis.plot(xTest, truth(xTest), label="Truth (true model)")
    axis.plot(xTest, yHat, label="Kernel Regression")

    axis.set_title(title)
    axis.set_xlabel(xlabel)
    axis.set_ylabel(ylabel)
    axis.legend()

    return axis


def bootstrap(x, y, B, kernel):
    """Performs a non-parametric bootstrap on the given dataset. Selects,
    with replacement, a subset of given features and labels, trains a kernel
    and creates new predictions on a (min(x), max(x)) range. Returns all made
    predictions.

    Parameters
    ----------
    x: `np.array`
        Features
    y: `np.array`
        Labels
    B: `int`
        Number of bootstrap iterations.

    Returns
    -------
    predictions: `np.array`
        Array each element of which is a set of predictions on a min(x)-max(x)
        range, i.e. each element are that bootstrap iterations predictions.
    percentile5: `np.array`
       Array each element of which is the 5th percentile of corresponding
       predictions element.
    percentile95: `np.array`
       Array each element of which is the 95th percentile of corresponding
       predictions element.
    """
    n = len(x)
    xTest = np.linspace(x.min(), x.max(), 100)
    predictions = np.zeros((B, len(xTest)))

    indices = np.arange(n)
    for i in range(B):
        idxs = np.random.choice(indices, size=n, replace=True)
        kernel.fit(x[idxs], y[idxs])
        predictions[i] = kernel.predict(xTest)

    return (predictions,
            np.percentile(predictions, 5, axis=0, interpolation="lower"),
            np.percentile(predictions, 95, axis=0, interpolation="higher"))


def A3c(kernelType, x, y, bestFit, B=300, title=""):
    """Bootstraps and estimates 5th and 95th percentile and then overplots it
    on data, true model and best fit estimate model.

    Parameters
    ---------
    kernelType: `str`
        Kernel type (``poly`` or `rbf`)
    x: `np.array`
        Features
    y: `np.array`
        Labels
    bestFit: `dict`
        A dictionary containing the minimal sampled cross validation error and
        the values of lambda and hyperparameter at that error.
    """
```

```
    fig, ax = plt.subplots()
    ax = A3b(kernelType, x, y, bestFit, axis=ax, title=title)

    kernel = KernelFactory.create(kernelType, bestFit['lambda'], bestFit['hyperparam'])
    predictions, percentile5, percentile95 = bootstrap(x, y, B, kernel)

    xTest = np.linspace(x.min(), x.max(), 100)
    ax.fill_between(xTest, percentile5, percentile95, alpha=0.3, color="gray")
    ax.plot(xTest, percentile95, color="darkgray", ls="--", alpha=0.5)
    ax.plot(xTest, percentile5, color="darkgray", ls="--", alpha=0.5)
    ax.set_ylim((y.min()-1, y.max()+1))

    return fig, ax


def A3e(bestPoly, bestRbf, n=1000, B=300):
    """Using the given kernel parameters fits olynomial and RBF kernels to
    the data, created according to the same truth, and calculates the mean
    difference of the squared errors of the kernels predictions via
    non-parametric bootstrap approach.

    Prints the 5th and 95th percentile of the confidence interval squared
    errors differences.

    Parameters
    ----------
    bestPoly: `dict`
        A dictionary containing the minimal sampled cross validation error and
        the values of lambda and hyperparameter at that error.
    bestRbf: `dict`
        A dictionary containing the minimal sampled cross validation error and
        the values of lambda and hyperparameter at that error.
    n: `int`, optional
        Number of newly generated data points, default: 1000.
    B: `int`, optional
        number of bootstrap iterations, default: 300.
    """
    x = np.random.uniform(size=n)
    y = truth(x) + np.random.normal(size=n)

    poly = KernelFactory.create("poly", bestPoly['lambda'], bestPoly['hyperparam'])
    rbf = KernelFactory.create("rbf", bestRbf['lambda'], bestRbf['hyperparam'])
    poly.fit(x, y)
    rbf.fit(x, y)

    sqErr = []
    indices = np.arange(n)
    for i in range(B):
        idxs = np.random.choice(indices, size=n, replace=True)
        predictPoly = poly.predict(x[idxs])
        predictRbf = rbf.predict(x[idxs])
        sqErr.append( np.mean((y[idxs]-predictPoly)**2 - (y[idxs]-predictRbf)**2) )

    percentile5 = np.percentile(sqErr, 5, axis=0, interpolation="lower")
    percentile95 = np.percentile(sqErr, 95, axis=0, interpolation="higher")

    print(f"Confidence interval difference: {percentile5} to {percentile95}")


def A3(n=30, foldSize=30, doPoly=True, doRBF=True, doA3e=False):
    """Problem A3 from a-d: creates data and labels based on truth and adds
    gaussian noise, performs a grid search for best regularization and
    hyperparameter values by minimizing the cross validation error, plots the
    fits, uses reported values to fit kernels accross the range of the given
    data, plots the kernel basis functions, the best fit kernels and boostraps
    5th and 95th percentile confidence intervals over the data range.

    Parameters
    ----------
    n: `int`, optional
        Number of data points to create, default: 30.
    foldSize: `int`, optional
        Cross validation set size, default: 30.
    doPoly: `bool`, optional
        Use polynomial kernel, default: True.
    doRBF: `bool`, optional
        Use RBF kernel, default: True.
    """
    x = np.random.uniform(size=n)
    y = truth(x) + np.random.normal(size=n)

    if doPoly:
        lambdas = np.arange(0.3, 0.7, 0.01)
        degrees = np.arange(0, 20, 1)
        samplesPoly, bestPoly = A3a("poly", x, y, foldSize, lambdas, degrees,
                                    xlabel="degree", title="Polynomial kernel")
        A3b("poly", x, y, bestPoly, title="Polynomial Kernel")
        A3c("poly", x, y, bestPoly, title="Polynomial Bootstrap (B=300) confidence intervals.")

    if doRBF:
        lambdas = np.arange(0.001, 0.6, 0.01)
        gammas = np.arange(0.1, 20, 0.25)
        samplesRdf, bestRdf = A3a("rbf", x, y, foldSize, lambdas, gammas,
                                  xlabel="gamma", title="RBF Kernel")
        A3b("rbf", x, y, bestRdf, title="RBF Kernel")
        A3c("rbf", x, y, bestRdf, title="RBF Bootstrap (B=300) confidence intervals.")

    if doA3e:
        A3e(bestPoly, bestRdf)

    plt.show()


def A3parallel(nprocs=None):
    """Runs A3 with 30 and 300 data points and 30 and 10 cross validation
    folding size in a parallel manner to amortize the total serial execution
    time.
    """
    args = [(30, 30, True, False), (30, 30, False, True),
            (300, 10, True, False), (300, 10, False, True),
```

```
            (300, 10, True, True, True)
            ]
    nprocs = len(args) if nprocs is None else nprocs
    with mp.Pool(nprocs) as p:
        p.starmap(A3, args)


if __name__ == "__main__":
    A3parallel()
```

k-means clustering

A4. Given a dataset $x_1, \ldots, x_n \in \mathbb{R}^d$ and an integer $1 \leq k \leq n$, recall the following k-means objective function

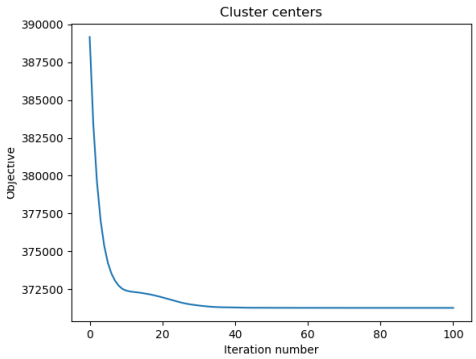
$$\min_{pi_1,\ldots,\pi_k} \sum_{i=1}^{k} \sum_{j \in \pi_i} ||x_j - \mu_i||_2^2$$

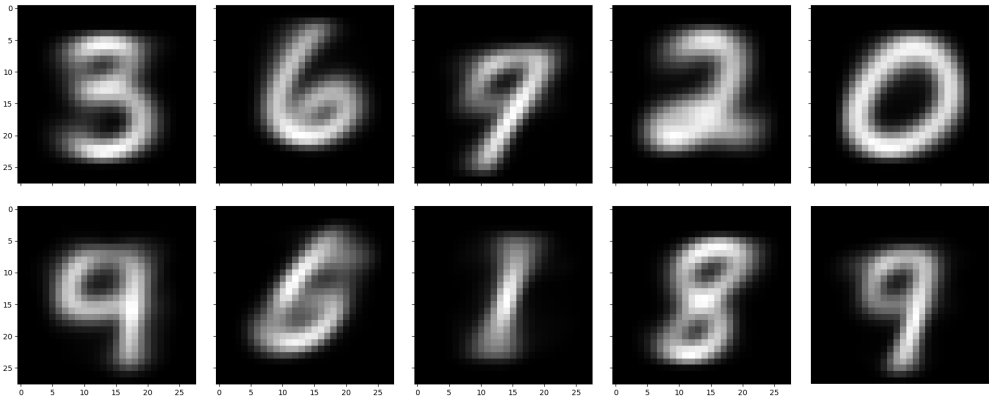
$$\mu_i = \frac{1}{\pi_i} \sum_{j \in \pi_i} x_j$$

Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, \ldots, n\}$. The objective is NP-hard to find a global minimizer of. Nevertheless Lloyd's algorithm, the commonly-used heuristic which we discussed in lecture, typically works well in practice.

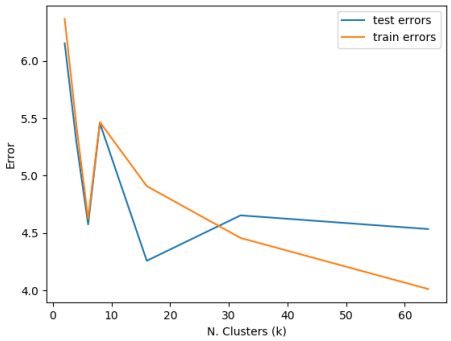
a. *[5 points]* Implement Lloyd's algorithm for solving the k-means objective. Do not use any off-the-shelf implementations, such as those found in scikit-learn. Include your code in your submission.

b. *[5 points]* Run the algorithm on the training dataset of MNIST with k=10, plotting the objective function as a function of the iteration number. Visualize (and include in your report) the cluster centers as a 28×28 image.





c. *[5 points]* For $k = \{2, 4, 8, 16, 32, 64\}$ run the algorithm on the training dataset to obtain centers $\{\mu_i\}_{i=1}^k$. If $\{(x_i, y_i)\}_{i=1}^n$ and $\{(x_i', y_i')\}_{i=1}^m$ denote the training and test sets, respectively, plot the training error $\frac{1}{n}\sum_{i=1}^n \min_{j=1,\ldots,k} ||\mu_j - x_i||_2^2$ and test error $\frac{1}{m}\sum_{i=1}^m \min_{j=1,\ldots,k} ||\mu_j - x_i'||_2^2$ as a function of k on the same plot.




```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST

np.random.seed(0)

def load_mnist_dataset(path="data/mnist_data/"):
    """Loads MNIST data located at path.

    MNIST data are 28x28 pixel large images of numbers.

    Parameters
    ----------
    path : `str`
        path to the data directory

    Returns
    -------
    train : `np.array`
        train data normalized to 1
    trainlabels : `np.array`
        train data labels
    test : `np.array`
        test data normalized to 1
    testLabels : `np.array`
        test data labels
    """
    mndata = MNIST(path)

    train, trainLabels = map(np.array, mndata.load_training())
    test, testLabels  = map(np.array, mndata.load_testing())

    train = train/255.0
    test = test/255.0

    return train, trainLabels, test, testLabels


def k_means_objective(clusters, centers):
    """Calculates the sum of distances of points in a cluster to the given
    cluster centers. This is the k-means objective, defined as:

        F(mu, C) = \sum^m || mu_j - x_j ||^2

    where mu are the cluster centers and x_j the point in that cluster.

    Parameters
    ----------
    clusters: `np.array`
        Clusters of points (each element a collection of points belonging to
        that cluster)
    centers: `np.array`
        Coordinates of centers of corresponding clusters.

    Returns
    -------
    objective: `float`
        K-means objective
    """
    dist = [np.sum(np.linalg.norm(c-p, axis=1)) for c, p in zip(centers, clusters)]
    return np.sum(dist)


def cluster_data(points, centers):
    """Calculates distance from each point to all given clusters and forms
    clusters by associatin points with its closest center.

    Parameters
    ----------
    points: `np.array`
        Points to cluster
    centers: `np.array`
```

```
        Cluster centers.

    Returns
    -------
    clusters: `np.array`
        Array each element of which is an cluster. A cluster is an array of
        points that bellonging to the same cluster.
    """
    nClusters = len(centers)
    # equivalent to: np.linalg.norm(points - centers[:, None], axis=2)
    # but for more clusters memory just runs out
    distances = [np.linalg.norm(points - center, axis=1) for center in centers]
    closestClusters = np.argmin(np.array(distances), axis=0)
    clusters = np.array([points[closestClusters == i] for i in range(nClusters)])
    return clusters


def calculate_centers(clusters):
    """Calculates centers of given clusters by calculating the mean of the
    individual coordinates of points in that cluster.

    Parameters
    ---------
    clusters: `np.array`
        An array in which each element is an array of points belonging to the
        cluster.

    Returns
    -------
    centers: `np.array`
        Cluster centers.
    """
    cluster_sizes = np.array([cluster.shape[0] for cluster in clusters])
    centers = np.array([np.mean(cluster, axis=0) for cluster in clusters])
    return centers


def loyds_alg(points, nClusters, tolerance=0.01, nIter=None):
    """Iteratively calculates centers and re-assinged clusters based on those
    centers until convergence is achieved. Intial centers are selected as
    random points from the given dataset. Convergence is achieved then the new
    center coordinate components maximal distance from the old center
    coordinates is smaller than tolerance. This is known as Loyd's algorithm
    for calculating k-means.

    Paramaters
    ----------
    points: `np.array`
        Array of points to cluster
    nClusters: `int`
        Number of clusters
    tolerance: `float`, optional
        If new centers move, cumulatively, by less than tolerance the iteration
        is terminated. Unless a specific number of iterations was given. By
        default: 0.01
    nIter: `int`, optional
        If given, tolerance is ignored and iterations are carried out for the
        given number of iterations.
    """
    # assign first centers to be random points in the dataset
    centers = points[np.random.permutation(np.arange(len(points)))[:nClusters]]
    clusters = cluster_data(points, centers)

    oldCenters, oldClusters = centers, clusters
    objectives = []
    converged, drOld, i = False, None, 0
    while not converged:
        centers = calculate_centers(clusters)
        clusters = cluster_data(points, centers)

        # worlds most complicated break-out logic
        dr = np.linalg.norm(centers - oldCenters)
        if drOld is None:
            print(dr)
            drOld = dr
        elif np.abs(drOld-dr) < tolerance:
            print(dr, "         ", drOld-dr)
            if nIter is not None:
                if i >= nIter:
                    converged = True
            else:
                converged = True
        else:
            print(dr, "         ", drOld-dr)
            drOld = dr

        objective = k_means_objective(clusters, centers)
        objectives.append(objective)

        oldCenters = centers
        oldClusters = clusters
        i+=1

    return centers, clusters, objectives


def plot_objectives(ax, objectives, xlabel="Iteration number", ylabel="Objective",
                    title="Cluster centers"):
    """Plots objectives.

    Paramters
    ---------
    ax: `matplotlib.pyplot.Axes`
        Axis to plot on.
    objectives: `np.array`
        Array of objective scores.
    xlabel: `str`, optional
        X label
    ylabel: `str`, optional
        Y label
```

```
    title: `str`, optional
        Axis title.
    """
    ax.plot(objectives)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    return ax


def plot_centers(axes, centers, title=""):
    """Plots centers as 28x28 images.

    Paramters
    ---------
    axes: `matplotlib.pyplot.Axes`
        Axes to plot on.
    centers: `np.array`
        Centers to plot.
    title: `str`, optional
        Axis title.
    """
    for ax, center in zip(axes.ravel(), centers):
        ax.imshow(center.reshape((28, 28)), cmap='gray')
        ax.set_title(title)
        plt.axis("off")
    return axes


def A4b(k=10, tolerance=0.01, nIter=100):
    """Runs Loyd's k-means algorithm on the MNIST test data-set for 100
    iterations, clustering the data into 10 clusters, and plots the objective
    function vs iteration number and the found centers.

    Parameters
    ----------
    k: `int`
        Number of clusters
    tolerance: `float`, optional
        Convergence tolerance, ignored if nIter is given.
    nIter: `int`, optional
        Number of iterations to carry out, defaults to 100.
    """
    test, testLabels, train, trainLabels = load_mnist_dataset()
    centers, clusters, objectives = loyds_alg(test, k, tolerance, nIter)

    fig1, axis1 = plt.subplots()
    plot_objectives(axis1, objectives)
    plt.show()

    fig, axes = plt.subplots(2, 5, figsize=(10, 25), sharex=True, sharey=True)
    plot_centers(axes, centers)
    plt.axis("off")
    plt.show()


def error(clusters, centers, nPoints=1):
    """Given clusters and centers calculates the total distance of points in
    that cluster to its center ("spread of points") and adds the least spread
    out cluster distances together into a total distance. Normalizes the total
    distance by the number of points.

    Paramters
    ---------
    clusters: `np.array`
        Array of clusters of points.
    centers: `np.array`
        Array of centers of clusters.
    nPoints: `int`, optional
        Normalization factor. Defaults to 1.

    Returns
    -------
    totDist: `float`
        Total sum of "spreads" of all the points in their respective clusters.
    """
    totDist = 0
    for center in centers:
        allDists = []
        for cluster in clusters:
            dr = np.linalg.norm(cluster-center, axis=1)
            allDists.append(np.sum(dr))
        totDist += min(allDists)
    return totDist/nPoints


def A4c(k=(2, 4, 6, 8, 16, 32, 64)):
    """Clusters MNIST test and train datasets into 2^n n=1,2,3,4,5,6 clusters
    and calculates the total error of the clustering as a function of the
    number of clusters. Plots the error.
    Clustering iterations are terminated when the total moved center distances
    are less than 0.01.

    Parameters
    ----------
    k: `tuple`
        Tuple of integers representing number of iterations.
    """
    test, testLabels, train, trainLabels = load_mnist_dataset()

    testErrors, trainErrors = [], []
    for nClusters in k:
        centers, clusters, objectives = loyds_alg(test, nClusters)
        testErrors.append(error(clusters, centers, len(test)))
        testClusters = cluster_data(train, centers)
        trainErrors.append(error(testClusters, centers, len(train)))

    plt.plot(k, testErrors, label="test errors")
    plt.plot(k, trainErrors, label="train errors")
    plt.xlabel("N. Clusters (k)")
```

```
    plt.ylabel("Error")
    plt.legend()
    plt.show()


if __name__ == "__main__":
    A4b()
    A4c()
```

Neural Networks for MNIST

A5. In Homework 1, we used ridge regression for training a classifier for the MNIST data set. Students who did problem B.2 also used a random feature transform. In Homework 2, we used logistic regression to distinguish between the digits 2 and 7. Students who did problem B.4 extended this idea to multinomial logistic regression to distinguish between all 10 digits. In this problem, we will use PyTorch to build a simple neural network classifier for MNIST to further improve our accuracy. We will implement two different architectures: a shallow but wide network, and a narrow but deeper network. For both architectures, we used to refer to the number of input features (in MNIST, $d = 28^2 = 784$), h_i to refer to the dimension of the i-th hidden layer and k for the number of target classes (in MNIST, $k = 10$).For the non-linear activation, use ReLU. Recall from lecture that

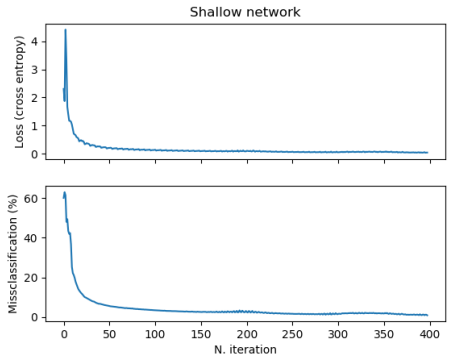
$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Weight Initialization
Consider a weight matrix $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$. Note that here m refers to the input dimension and n to the output dimension of the transformation $Wx + b$. Define $\alpha = 1/\sqrt{m}$. Initialize all your weight matrices and biases according to $Unif(-\alpha, \alpha)$.
Training
For this assignment, use the Adam optimizer from torch.optim. Adam is a more advanced form of gradient descent that combines momentum and learning rate scaling. It often converges faster than regular gradient descent. You can use either Gradient Descent or any form of Stochastic Gradient Descent. Note that you are still using Adam, but might pass either the full data, a single datapoint or a batch of data to it. Use cross entropy for the loss function and ReLU for the non-linearity.
Implementing the Neural Networks

a. *[10 points]* Let $W_0 \in \mathbb{R}^{h \times d}$, $b_0 \in \mathbb{R}^h$, $W_1 \in \mathbb{R}^{k \times h}$, $b_1 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the wide, shallow network can be formulated as:

$$\mathbb{F}_1(x) = W_1\sigma(W_0x + b_0) + b + 1$$

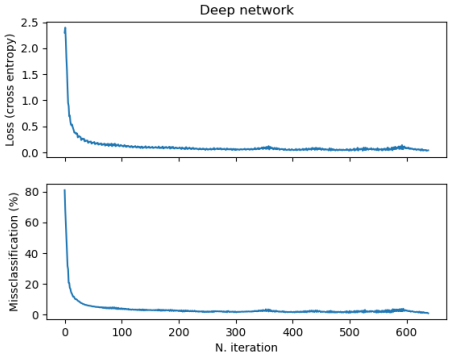
Use $h = 64$ for the number of hidden units and choose an appropriate learning rate. Train the network until it reaches 99% accuracy on the training data and provide a training plot (loss vs. epoch). Finally evaluate the model on the test data and report both the accuracy and the loss.
Accuracy (train): 99.015
Loss (train): 0.03885
Accuracy (test) 96.02
Loss (test): 0.17849



b. *[10 points]* Let $W_0 \in \mathbb{R}^{h_0 \times d}$, $b_0 \in \mathbb{R}^{h_0}$, $W_1 \in \mathbb{R}^{h_1 \times h_0}$, $b_1 \in \mathbb{R}^{h_1}$, $W_2 \in \mathbb{R}^{k \times h_2}$, $b_2 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the network can be formulated as:

$$\mathbb{F}_2(x) = W_2\sigma(W_1\sigma(W_0x + b_0) + b_1) + b_2$$

Use $h_0 = h_1 = 32$ and perform the same steps as in part a.
Accuracy (train): 99.123
Loss (train): 0.04016
Accuracy (test) 96.4
Loss (test): 0.20875



c. *[5 points]* Compute the total number of parameters of each network and report them. Then compare the number of parameters as well as the test accuracy the networks achieved. Is one of the approaches (wide,shallow vs narrow, deeper) better than the other? Give an intuition for why or why not.
The total number of parameters is the total number of elements in all layers. For shallow net for example we have zeroth layer with weights in a $d \times h$ where $d = 784$ and $h = 64$ and offsets with dimensions 1-by-64. The next layer has 64-by-10 weights and 1-by-10 offsets. Therefore there are $784 \cdot 64 + 64 + 64 \cdot 10 + 10 = 50890$ parameters. Same calculation holds for the deep neural net which as $k = 26506$ parameters.

The test accuracy are very similar but the deep network has significantly less parameters so we should probably pick that one. Bayesian information criterion is usually used to asses the model performance, with lower BIC preferred. BIC can be calculated as:

$$BIC = k \ln n - 2 \ln \widehat{L}$$

where n is the number of samples, k number of parameters and $\widehat{L}$ the maximized value of the likelihood function of the model, i.e. the test accuracy for best fitting trained model. There are 10 000 images in the test data set, this is our n, and the number of parameters we calculated above.
Putting in the numbers we get that for $BIC_a = 468714.3$ and $BIC_b = 244129.4$. This confirms the idea that the deep network out-preforms the shallow network. I don't think this is a surprise because applying the non-linear activator twice would be better, purely intuitively, at discerning the smaller differences between numbers such as 3 and 8 for example. In a single, more linear-like, neural network these differences will be closer in the projected space than in the deep network.

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
import torch
import torch.nn as nn

np.random.seed(0)
torch.manual_seed(0)

def load_mnist_dataset(path="data/mnist_data/"):
    """Loads MNIST data located at path.

    MNIST data are 28x28 pixel large images of numbers.

    Parameters
    ----------
    path : `str`
        path to the data directory

    Returns
    -------
    train : `torch.tensor`
        train data normalized to 1
    trainlabels : `torch.tensor`
        train data labels
    test : `torch.tensor`
        test data normalized to 1
    testLabels : `torch.tensor`
        test data labels
    """
    mndata = MNIST(path)

    train, trainLabels = map(torch.tensor, mndata.load_training())
    test, testLabels  = map(torch.tensor, mndata.load_testing())

    train = train/255.0
    test = test/255.0

    return (train.type(torch.DoubleTensor), trainLabels.long(),
        test.type(torch.DoubleTensor), testLabels.long())


def calculate_error(model, x, y):
    """Given a model, features and labels calculates the models missclassification
    error in percentage.

    Parameters
    ----------
    model: `func`
        Model, a function that takes in features and returns labels.
    x: `torch.tensor`
        Features
    y: `torch.tesnor`
        Labels

    Returns
    -------
    error: `float`
        Missclassification error.
    """
    yHat = model(x)
    predictions = torch.argmax(yHat, dim=1)
    return float(100*(predictions != y).float().mean())


def train(model, optimizer, lossFunc, x, y, batchSize, tolerance=1):
    """Given a model, optimizer, desirdata and batch size
    trains the model.

    Parameters
    ----------
    model: `func`
        Model, a function that takes in features and returns labels.
    optimizer: `class`
        Optimizer, one of torch.optim classes (e.g. torch.optim.Adam)
    lossFunc: `class`
        Loss function, one of torch.nn classes (e.g. torch.nn.CrossEntropyLoss)
    x: `torch.tensor`
        Features
    y: `torch.tensor`
        Labels
    batchSize: `int`
        Size of batches used in training.
    tolerance: `float`
        When error becomes smaller than tolerance, iterations are terminated.

    Returns
    -------
    errors: `list`
        Errors per epoch
    losses: `list`
        Loss per epoch

    Notes
    -----
    Model is trained for 1000 epoch. Error and loss for each epoch is recorded.
    """
    indices = np.arange(len(x))
    nIter, converged = 0, False
    errors, losses = [], []
    for i in range(1000):
        indexList = np.random.permutation(np.arange(len(x)))
        batches = np.split(indices, batchSize)
        for batch in batches:
            data = x[batch]
            labels = y[batch]

            fitted = model(data)

            optimizer.zero_grad()
```

```
            loss = lossFunc(fitted, labels)
            loss.backward()
            optimizer.step()

            newError = calculate_error(model, x, y)
            errors.append(newError)
            losses.append(float(loss))

            if newError < tolerance:
                converged = True
                break

        if converged:
            break

    return errors, losses


def define_model(w0, b0, w1, b1, w2=None, b2=None, sigma=nn.ReLU(), which="a"):
    """A closure that returns a model with the given layer weights and offsets.
    Defines two models "a" and "b" (aka shallow and deep network) for the two
    networks defined in the problem. Models take in a single input, the
    features.

    Parameters
    ----------
    w*: `torch.tensor`
       Layer weights, number indicates the layer depth. Acceptable depths are
       from 0 to 2 (e.g. w0, w1, w2).
    b*: `torch.tensor`
       Layer offsets, number indicates layer depth. Acceptable depths are
       from 0 to 2 (e.g. b0, b1, b2).
    sigma: `func`
        Activation function, e.g. `torch.nn.ReLu()`
    which: `str`, optional
        Which model to return, accepts ``a`` or ``b``. Default: ``a``

    Returns
    -------
    model: `func`
        Singgle argument function, the model.
    """
    def A5a_model(x):
        """Two layer model: ReLU(W0 X + b0) W2 + B2 """
        inner = x@w0 + b0
        return sigma(inner) @ w1 + b1

    def A5b_model(x):
        """Three layer model: ReLu( W1 (ReLU(W0 X + b0) W2 + B2) + B1) W2 +B2"""
        innest = x@w0 + b0
        inner = sigma(innest) @ w1 + b1
        return sigma(inner) @ w2 + b2

    if which == "a":
        return A5a_model
    elif which == "b":
        return A5b_model
    else:
        raise AttributeError("You missed!, How could you miss!? He was three feet "
                             "in front of you!\n\t - Mushu in a snowy pass, 1998")

def plot_paths(losses, errors):
    """Plots given losses and errors as a function of epoch.

    Paramters
    ---------
    losses: `list`
        Loss per epoch
    errors: `list`
        Error per epoch
    """
    fig, axes = plt.subplots(2,1, sharex=True)

    x = np.arange(len(losses))

    axes[0].plot(x, losses)
    axes[0].set_ylabel("Loss (cross entropy)")

    axes[1].plot(x, errors)
    axes[1].set_ylabel("Missclassification (%)")
    axes[1].set_xlabel("N. iteration")

    plt.show()


def A5a(learnRate=0.05, batchSize=6):
    """Defines weights and offsets of a 2 layered neural network, trains it,
    calculates train and test accuracy and losses and plots training losses and
    errors per epoch

    Parameters
    ----------
    learnRate: `float`, optional
        Learning rate, default 0.05
    batchSize: `int`, optional
        Training batch size, default 6
    """
    n, d, h, k = 28, 28**2, 64, 10

    tDim, lDim = 1/n, 1/np.sqrt(h)
    w0 = torch.DoubleTensor(d, h).uniform_(-tDim, tDim).requires_grad_()
    b0 = torch.DoubleTensor(1, h).uniform_(-tDim, tDim).requires_grad_()
    w1 = torch.DoubleTensor(h, k).uniform_(-lDim, lDim).requires_grad_()
    b1 = torch.DoubleTensor(1, k).uniform_(-lDim, lDim).requires_grad_()
    breakpoint()

    trainData, trainLabels, testData, testLabels = load_mnist_dataset()

    optimizer = torch.optim.Adam([w0, b0, w1, b1], lr=learnRate)
    model = define_model(w0, b0, w1, b1)
```

```
    loss = nn.CrossEntropyLoss()

    errors, losses = train(model, optimizer, loss, trainData, trainLabels,
                           batchSize)

    # calculate loss and accuracy on testing set
    yHat = model(testData)
    testLoss = loss(yHat, testLabels)

    print(f"Accuracy (train): {100 - errors[-1]}")
    print(f"Loss (train): {losses[-1]}")

    print(f"Accuracy (test) {100 - calculate_error(model, testData, testLabels)}")
    print(f"Loss (test): {float(testLoss)}")

    plot_paths(losses, errors)


def A5b(learnRate=0.05, batchSize=6):
    """Defines weights and offsets of a 3 layered neural network, trains it,
    calculates train and test accuracy and losses and plots training losses and
    errors per epoch

    Parameters
    ----------
    learnRate: `float`, optional
        Learning rate, default 0.05
    batchSize: `int`, optional
        Training batch size, default 6
    """

    n, d, h0, h1, k = 28, 28**2, 32, 32, 10

    tDim, l0Dim, l1Dim = 1/n, 1/np.sqrt(h0), 1/np.sqrt(h1)
    w0 = torch.DoubleTensor(d, h0).uniform_(-tDim, tDim).requires_grad_()
    b0 = torch.DoubleTensor(1, h0).uniform_(-tDim, tDim).requires_grad_()
    w1 = torch.DoubleTensor(h0, h1).uniform_(-l0Dim, l0Dim).requires_grad_()
    b1 = torch.DoubleTensor(1, h1).uniform_(-l0Dim, l0Dim).requires_grad_()
    w2 = torch.DoubleTensor(h1, k).uniform_(-l1Dim, l1Dim).requires_grad_()
    b2 = torch.DoubleTensor(1, k).uniform_(-l1Dim, l1Dim).requires_grad_()

    trainData, trainLabels, testData, testLabels = load_mnist_dataset()

    optimizer = torch.optim.Adam([w0, b0, w1, b1, w2, b2], lr=learnRate)
    model = define_model(w0, b0, w1, b1, w2, b2, which="b")
    loss = nn.CrossEntropyLoss()

    errors, losses = train(model, optimizer, loss, trainData, trainLabels,
                           batchSize)

    # calculate loss and accuracy on testing set
    yHat = model(testData)
    testLoss = loss(yHat, testLabels)

    print(f"Accuracy (train): {100 - errors[-1]}")
    print(f"Loss (train): {losses[-1]}")

    print(f"Accuracy (test) {100 - calculate_error(model, testData, testLabels)}")
    print(f"Loss (test): {float(testLoss)}")

    plot_paths(losses, errors)


if __name__ == "__main__":
    A5a()
    print()
    A5b()
```

PCA

Let's do PCA on MNIST dataset and reconstruct the digits in the dimensionality-reduced PCA basis. You will actually compute your PCA basis using the training dataset only, and evaluate the quality of the basis on the test set, similar to the k-means reconstructions of above. Because 50,000 training examples are size 28×28 so begin by flattening each example to a vector to obtain $X_{\text{train}} \in \mathbb{R}^{50,000 \times d}$ and $X_{\text{test}} \in \mathbb{R}^{10,000 \times d}$ for $d := 784$
A6. Let $\mu \in \mathbb{R}^d$ denote the average of the training examples in X_{train}, i.e., $\mu = \frac{1}{d} X_{\text{train}}^T$. Now let $\sum (X_{\text{train}} - \mathbf{1}\mu^T)^T (X_{\text{train}} - \mathbf{1}\mu^T)/50000$ denote the sample covariance matrix of the training examples, and let $\sum = UDU^T$ denote the eigenvalue decomposition of $\sum$

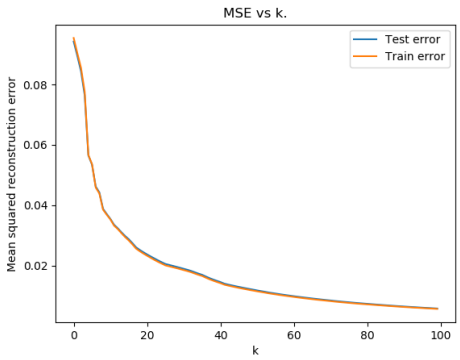
a. *[2 points]* If λ_i denotes the $i-th$ largest eigenvalue of $\sum$, what are the eigenvalues $\lambda_1, \lambda_2, \lambda_{10}, \lambda_{30}$, and λ_{50}? What is the sum of eigenvalues $\sum_{i=1}^{d} \lambda_i$?
1th eigenvalue: 5.1168
2th eigenvalue: 3.7413
10th eigenvalue: 1.2427
30th eigenvalue: 0.3642
50th eigenvalue: 0.1697
Sum of eigenvalues: 52.7250

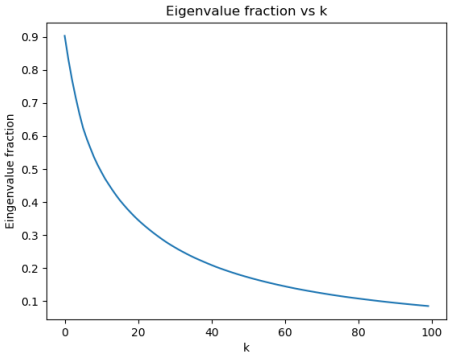
b. *[5 points]* Any example $x \in \mathbb{R}^d$ (including those from either the training or test set) can be approximated using just μ and the first k eigenvalue, eigenvector pairs, for any $k = 1, 2, \ldots, d$. For any k, provide a formula for computing this approximation.

c. *[5 points]* Using this approximation, plot the reconstruction error from $k = 1$ to 100 (the X-axis is k and the Y-axis is the mean-squared error reconstruction error) on the training set and the test set (using the μ and the basis learned from the training set). On a separate plot, plot

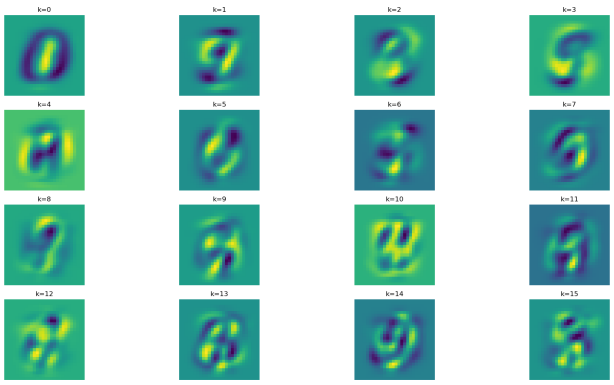
$$1 - \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$$

from $k = 1$ to 100.

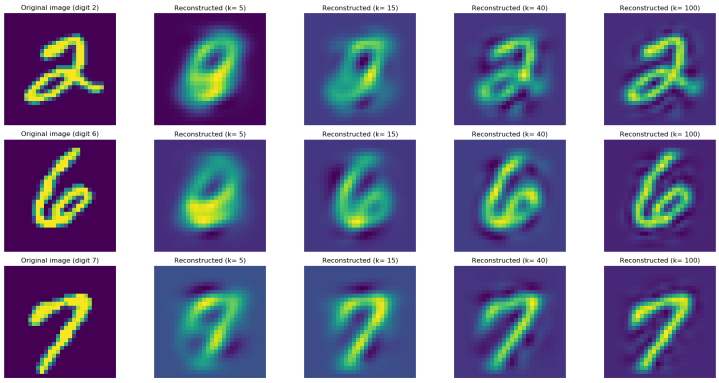




d. *[3 points]* Now let us get a sense of what the top PCA directions are capturing. Display the first 10 eigenvectors as images, and provide a brief interpretation of what you think they capture.
I think they mainly focus on areas where most numbers overlap. For example only zero will fill a circle in the image, while all other numbers will have at least one line cross the middle of the image. So it's not surprising the first PCA components is very much focused on the middle. The following components begin to fill out arches of the numbers (e.g. where top of the 9, 5 or 3 would be). Zero is then constructed as a combination of all outer arches etc. In any case, it looks as if PCA components focus more on areas where the numbers "aren't", on the differences between numbers rather than their common components. The common component then is reconstructed by some linear combination of the oth basis vector.



e. *[3 points]* Finally, visualize a set of reconstructed digits from the training set for different values of k. In particular provide the reconstructions for digits 2,6,7 with values $k = 5, 15, 40, 100$ (just choose an image from each digit arbitrarily). Show the original image side-by-side with its reconstruction. Provide a brief interpretation, in terms of your perceptions of the quality of these reconstructions and the dimensionality you used.
I think reconstruction after 40 used eigenvectors is rather really good. The 100




```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
import time


np.random.seed(0)


def load_mnist_dataset(path="data/mnist_data/"):
    """Loads MNIST data located at path.

    MNIST data are 28x28 pixel large images of numbers.

    Parameters
    ----------
    path : `str`
        path to the data directory
```

```
    Returns
    -------
    train : `torch.tensor`
        train data normalized to 1
    trainlabels : `torch.tensor`
        train data labels
    test : `torch.tensor`
        test data normalized to 1
    testLabels : `torch.tensor`
        test data labels
    """
    mndata = MNIST(path)

    train, trainLabels = map(np.array, mndata.load_training())
    test, testLabels  = map(np.array, mndata.load_testing())

    train = train/255.0
    test = test/255.0

    return train, trainLabels, test, testLabels


def calculate_errors(x, eigenvectors=None, transMatrix=None):
    """Calculates mean square error of PCA prediction.

    Paramters
    ---------
    x: `np.array`
        Features
    Vk: `np.array`, optional
        Eigenvectors
    transMatrix: `np.array`, None
        Transformation matrix, the dot product of eigenvectors with themselves
        trainsposed.

    Returns
    -------
    error: `float`
        Means square error of reconstruction.
    """
    if transMatrix is None:
        if eigenvectors is None:
            raise AttributeError("Need to supply Vk!")
        transMatrix = np.dot(eigenvectors, eigenvectors.T)
    return np.mean((x - np.dot(x, transMatrix))**2)


def plot_eigen_fraction(k, frac):
    """Plots eingenvalue fraction as a function of the total number of
    eingenvectors found during decomposition.

    Parameters
    ----------
    k: `int`
        Number of fitted eigenvectors
    frac: `float`
        Fraction of eigenvalues over total eigenvalue sum.
    """
    fig, ax = plt.subplots()
    ax.plot(k, frac)
    ax.set_xlabel("k")
    ax.set_ylabel("Eingenvalue fraction")
    ax.set_title("Eigenvalue fraction vs k")
    plt.show()


def plot_errors(k, test, train):
    """Plots test and train error as a function of the number of eigenvectors
    used in reconstruction.

    Parameters
    ----------
    k: `int`
        Number of used eigenvectors
    test: `np.array`
        Test error
    train: `np.array`
        Train error
    """
    fig, ax = plt.subplots()
    ax.plot(k, test, label="Test error")
    ax.plot(k, train, label="Train error")
    ax.set_xlabel("k")
    ax.set_ylabel("Mean squared reconstruction error")
    ax.set_title("MSE vs k.")
    ax.legend()
    plt.show()


def plot_n_eigenvectors(n, eigenvectors, nXaxes=2, nYaxes=5):
    """Plots first n eigenvectors

    Parameters
    ---------
    n: `int`
        Number of vectors to plot
    eigenvectors: `np.array`
        Eigenvectors
    nXaxes: `int`, optional
        Number of figure axes in the x direction
    nYaxes: `int`, optional
        Number of figure axes in the y direction
    """
    fig, axes = plt.subplots(nXaxes, nYaxes)

    for ax, k, eigVec in zip(axes.ravel(), range(n), eigenvectors.T):
        ax.imshow(eigVec.reshape((28, 28)))
        ax.set_title(f"k={k}")
        ax.axis("off")
```

```
    plt.show()


def plot_pca(x, y, eigenvectors, mu, digits =(2, 6, 7), ks=(5, 15, 40, 100)):
    """Plots the original digits and their reconstruction for different number
    of used eigenvectors.

    Parameters
    ----------
    x: `np.array`
        Features
    y: `np.array`
        Labels
    eigenvectors: `np.array`
        Eigenvectors
    mu: `np.array`
        Fitted mu.
    eigenvectors: `np.array`
        Eigenvectors
    digits: `tuple`
        Digits to plot
    ks: `tuple`
        Tuple of integers declaring how many eigenvectors should be used in
        reconstruction.
    """
    fig, axes = plt.subplots(len(digits), len(ks)+1)
    if len(digits) == 1:
        axes = np.array([axes])

    idxDigits = [np.where(y==digit)[0][0] for digit in digits]

    for yax, digit, idxDigit in zip(axes[:, 0], digits, idxDigits):
        yax.imshow(x[idxDigit].reshape((28, 28)))
        yax.set_title(f"Original image (digit {digit})")
        yax.axis("off")

    for yax, digit, idxDigit in zip(axes[:, 1:], digits, idxDigits):
        for xax, k in zip(yax, ks):
            Vk = eigenvectors[:, :k]
            reconstruction = np.dot(Vk, np.dot(Vk.T, (x-mu.T)[idxDigit])).reshape((784, 1))
            reconstruction += mu
            xax.imshow(reconstruction.reshape((28, 28)))
            xax.set_title(f"Reconstructed (k= {k})")
            xax.axis("off")

    plt.show()


def pca():
    """Preforms PCA on MNIST dataset.

    Calculates prints some eigenvalues, prints the sum of all eigenvalues.
    Plots first 25 eigenvectors.
    Plots eigenvalue fraction.
    Calculates the test and train errors for reconstructions up to first 100
    eigenvectors. Plots them.
    Reconstructs certain digits for varying number of used eigenvectors. Plots
    them.
    """
    train, trainLabels, test, testLabels = load_mnist_dataset()

    n, d = train.shape
    I = np.ones((n, 1))

    mu = np.dot(train.T, I)/n
    sigElem = train - np.dot(I, mu.T)
    sigma = np.dot(sigElem.T, sigElem)/n

    eigenvalues, eigenvectors = np.linalg.eigh(sigma)
    eigenvalues = eigenvalues[np.argsort(-1 * eigenvalues)]
    eigenvectors = eigenvectors[:, np.argsort(eigenvalues)]

    totEigenSum = np.sum(eigenvalues)

    trainErrors, testErrors, eigenRatios = [], [], []
    eigenSum, k = 0, np.arange(100)
    for i in k:
        Vk = eigenvectors[:, :(i+1)]
        transMatrix = np.dot(Vk, Vk.T)
        trainErrors.append(calculate_errors(train, transMatrix=transMatrix))
        testErrors.append(calculate_errors(test, transMatrix=transMatrix))
        eigenSum += eigenvalues[i]
        eigenRatios.append(1 - (eigenSum/totEigenSum))

    for i in (1, 2, 10, 30, 50):
        print(f"{i}th eigenvalue: {eigenvalues[i-1]}")
    print(f"Sum of eigenvalues: {totEigenSum}")

    plot_n_eigenvectors(16, eigenvectors, nXaxes=4, nYaxes=4)
    plot_eigen_fraction(k, eigenRatios)
    plot_errors(k, trainErrors, testErrors)
    plot_pca(train, trainLabels, eigenvectors, mu)


if __name__ == "__main__":
    pca()
```