Homework #1 A

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

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

Short Answer and "True or False" Conceptual questions

A.0 The answers to these questions should be answerable without referring to external materials.

a. *[2 points]* In your own words, describe what bias and variance are? What is bias-variance tradeoff?
Bias is the error we incur by assuming a model, since models don't necessarily describe the underlying truth they will attempt to "rephrase" it in terms of assumptions and/or approximations made in the model itself. Variance is the error that comes about due to the spread in our data. At a fine enough level all processes will have some amount of variance. Fitting a model to that data will produce an estimator of measured dataset. Remeasuring and refitting a model might produce a slightly, or significantly, different estimator due to the variance between measured data points. That slight difference in expected estimator is the model variance.
The more complex the model we use the more of intricate details of the underlying truth it can capture. This leads to low bias but also opens the doors to high variance if the measured dataset varies a lot between measurements. On the other hand, having a simple model might reduce the complexity and be able to better ignore the variance in the data but once fitted might not correspond with the truth. The inability to chose arbitrarily complex model without having the variance explode or, vice-versa, choosing arbitrarily simple model while not having bias explode is called bias-variance trade-off.

b. *[2 points]* What happens to bias and variance when the model complexity increases/decreases?
See above. Bias tends to reduce with model complexity but variance tends to increase.

c. *[1 points]* True or False: The bias of a model increases as the amount of training data available increases.
False. Bias is related to underlying inability of a model to describe the truth. That statement holds irregardless of the amount of data we have. The fit gets better because the variance of the model will reduce the more data we have to train on.

d. *[1 points]* True or False: The variance of a model decreases as the amount of training data available increases.
True. See above, variance decreases.

e. *[1 points]* True or False: A learning algorithm will generalize better if we use less features to represent our data.
T/F. Generalize with respect to what? Modelling broken spaghetti length that correctly predicts atmospheric pressure over Antarctica for 2 months of 2020 is general but not useful.

f. *[2 points]* To get better generalization, should we use the train set or the test set to tune our hyperparameters?
Always generalize on the test set. We're told it's a fact of life and I don't want to get yelled at by the professors.

g. *[1 points]* True or False: The training error of a function on the training set provides an overestimate of the true error of that function.
False. Training error will always be less than the test error. True error is estimated on test data set, not on the training data set.

Maximum Likelihood Estimation (MLE)

A.1. You're a Reign FC fan, and the team is five games into its 2018 season. The number of goals scored by the team in each game so far are: $[2, 0, 1, 1, 2]$. Let's call these scores $x_1, ..., x5$. Based on your (assumed iid) data, you'd like to build a model to understand how many goals the Reign are likely to score in their next game. You decide to model the number of goals scored per game using a Poisson distribution. The Poisson distribution with parameter λ assigns every non-negative integer $x = 0, 1, 2, ...$ a probability given by

$$P(x|\lambda) = \frac{\lambda^x}{x!}e^{-\lambda}$$

So, for example, if $\lambda = 1.5$, then the probability that the Reign score 2 goals in their next game is $e^{-1.5} \cdot 1.522! \approx 0.25$. To check your understanding of the Poisson, make sure you have a sense of whether raising λ will mean more goals in general, or fewer.

a. *[5 points]* Derive an expression for the maximum-likelihood estimate of the parameter λ governing the Poisson distribution, in terms of your goal counts $x_1, ..., x_5$. (Hint: remember that the log of the likelihood has the same maximum as the likelihood function itself.)

$$\begin{aligned}
\widehat{\lambda}_{MLE} &= \operatorname*{argmax}_{\lambda} L_n(\lambda) = \operatorname*{argmax}_{\lambda} \prod_{i=1}^{n} P(x_i|\lambda) \\
&= \operatorname*{argmax}_{\lambda} \prod_{i=1}^{n} \frac{\lambda^{x_i}}{x_i!}e^{\lambda} = \operatorname*{argmax}_{\lambda} \sum_{i=1}^{n} \ln \frac{\lambda^{x_i}}{x_i!}e^{\lambda} \\
&= \operatorname*{argmax}_{\lambda} \sum_{i=1}^{n} \left[\ln \lambda^{x_i} - \ln \frac{1}{x_i!} - \ln e^{\lambda}\right] \\
&= \operatorname*{argmax}_{\lambda} \sum_{i=1}^{n} [x_i \ln \lambda + \ln x_i! - \lambda] \\
&= \operatorname*{argmax}_{\lambda} \left[\ln \lambda \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} \ln x_i! - n\lambda\right]
\end{aligned}$$

Finding the $\operatorname{argmax}_{\lambda}$

$$\begin{aligned}
0 &= \frac{d}{d\lambda}\left[\ln \lambda \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} \ln x_i! - n\lambda\right] \\
0 &= \frac{1}{\lambda}\sum_{i=1}^{n} x_i + 0 + n \\
\lambda &= \frac{1}{n}\sum_{i=1}^{n} x_i = \bar{X} = \frac{2+0+1+1+2}{5} = 1.2 \\
&\rightarrow P(x|\lambda = 1.2) = \frac{1.2^x}{x!}e^{-1.2}
\end{aligned}$$

b. *[5 points]* Suppose the team scores 4 goals in its sixth game. Derive the same expression for the estimate of the parameter λ as in the prior example, now using the 6 games $x_1, ..., x_5, x_6 = 4$.
See above, lambda will be the average of X, so $\widehat{\lambda}_{MLE} = 2$.

c. *[5 points]* Given the goal counts, please give numerical estimates of λ after 5 and 6 games.
See above, 1.2 and 2 respectively.

A.2.*[10 points]* In World War 2, the Allies attempted to estimate the total number of tanks the Germans had manufactured by looking at the serial numbers of the German tanks they had destroyed. The idea was that if there were n total tanks with serial numbers $1, ..., n$ then its reasonable to expect the observed serial numbers of the destroyed tanks constituted a uniform random sample (without replacement) from this set. The exact maximum likelihood estimator for this so-called German tank problem is non-trivial and quite challenging to work out (try it!). For our homework, we will consider a much easier problem with a similar flavor. Let $x_1, ..., x_n$ be independent, uniformly distributed on the continuous domain $[0, \theta]$ for some θ. What is the Maximum likelihood estimate for θ
Repeat A.1. but with the following PDF:

$$P(X|\theta) = \begin{cases} \frac{1}{x_n - x_1} & \text{for } x_1 \leq x_i \leq x_n \\ 0 & \text{otherwise} \end{cases}$$

in which we can substitute given information $x_1 = 0$ and $x_n = \theta$ and rewrite

$$\begin{aligned} P(X|\theta) &= \begin{cases} \frac{1}{\theta} & \text{for } 0 \leq x_i \leq \theta \\ 0 & \text{otherwise} \end{cases} \\ &= \frac{1}{\theta}\mathbf{1}\{x \in [0, \theta]\} \end{aligned}$$

Likelihood and log-likelihood are given by:

$$\begin{aligned} L_n(X|\theta) &= \prod_{i=1}^{n} \frac{1}{\theta}\mathbf{1}\{x \in [0, \theta]\} \\ &= \frac{1}{\theta^n} \\ l_n(X|\theta) &= \ln L_n(X|\theta) = -n \ln \theta \end{aligned}$$

since normalization is given by θ. MLE is then found by

$$\begin{aligned} \widehat{\theta}_{MLE} &= \underset{\theta}{\text{argmax}}\, l_n(X|\theta) = \underset{\theta}{\text{argmax}}\, -n \ln \theta \\ 0 &= \frac{d}{d\theta}(-n \ln \theta) = \frac{-n}{\theta} \\ \theta &= -n \end{aligned}$$

Reversing the logic and looking at the negative log-likelihood $\frac{d}{d\theta} - l_n = \frac{n}{\theta}$ it is apparent the smaller the θ the larger the negative log-likelihood. So the negative log-likelihood $-l_n$ is minimized for the x_n closest to θ. It should be now apparent that log-likelihood is therefore maximized at $\theta_{MLE} = max(x_i \leq \theta)$.

A.3. Suppose we have N labeled samples $S = (x_i, y_i)_{i=1}^N$ drawn i.i.d. from an underlying distribution D. Suppose we decide to break this set into a set S_{train} of size N_{train} and a set S_{test} of size N_{test} samples for our training and test set, so $N = N_{\text{train}} + N_{\text{test}}$, and $S = S_{\text{train}} \cup S_{\text{test}}$. Recall the definition of the true least squares error off:

$$(f) = E_{(x,y)\approx D}[(f(x) - y)2]$$

where the subscript $(x, y) \approx D$ makes clear that our input-output pairs are sampled according to D. Our training and test losses are defined as:

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,y)\in S_{\text{train}}} (f(x) - y)^2$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,y)\in S_{\text{test}}} (f(x) - y)^2$$

We then train our algorithm (for example, using linear least squares regression) using the training set to obtain $\hat{f}$

a. *[3 points]* (bias: the test error) For all fixed f (before we've seen any data) show that

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] = \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \epsilon(f)$$

Use a similar line of reasoning to show that the test error is an unbiased estimate of our true error for $\hat{f}$. Specifically, show that:

$$\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(\hat{f}) = \widehat{\epsilon}(\hat{f})$$

b. *[4 points]* (bias: the train/dev error) Is the above equation true (in general) with regards to the training loss? Specifically, does $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\hat{f})] = \mathbb{E}_{\text{train}}[\epsilon(\hat{f})]$? If so, why? If not, give a clear argument as to where your previous argument breaks down.

c. *[8 points]* Let $F = (f_1, f_2, ...)$ be a collection of functions and $\hat{f}_{\text{train}}$ minimize the training error such that $\widehat{\epsilon}_{\text{train}}(\hat{f}_{\text{train}}) \leq \widehat{\epsilon}_{\text{train}}(f) \, \forall \, f \in F$. Show that

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\hat{f}_{\text{train}})] \leq \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\hat{f}_{\text{train}})]$$

(Hint: note that

$$\begin{aligned} E_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\hat{f}_{\text{train}})] = & \\ &= \sum f \in F \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(f) \mathbf{1} \hat{f}_{\text{train}} = f] \\ &= \sum f \in F \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] \mathbb{E}_{\text{train}}[\mathbf{1} \hat{f}_{\text{train}} = f] \\ &= \sum f \in F \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] \mathbb{P}_{\text{train}}(\hat{f}_{\text{train}} = f) \end{aligned}$$

where the second equality follows from the independence between the train and test set.)

Polynomial Regression

A.4 *[10 points]* Recall that polynomial regression learns a function $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_d x^d$. In this case, d represents the polynomial's degree. We can equivalently write this in the form of a linear model

$$h_\theta(x) = \theta_0\phi_0(x) + \theta_1\phi_1(x) + \theta_2\phi_2(x) + \ldots + \theta_d\phi_d(x).$$

using the basis expansion that $\phi_j(x) = x^j$. Notice that, with this basis expansion, we obtain a linear model where the features are various powers of the single univariate x. We're still solving a linear regression problem,but are fitting a polynomial function of the input. Implement regularized polynomial regression in polyreg.py. You may implement it however you like, using gradient descent or a closed-form solution. However, I would recommend the closed-form solution since the datasets are small; for this reason, we've included an example closed-form implementation of linear regression inlin regclosedform.py (you are welcome to build upon this implementation, but make CERTAIN you under-stand it, since you'll need to change several lines of it). You are also welcome to build upon your implementation from the previous assignment, but you must follow the API below. Note that all matrices are actually 2D numpy arrays in the implementation.

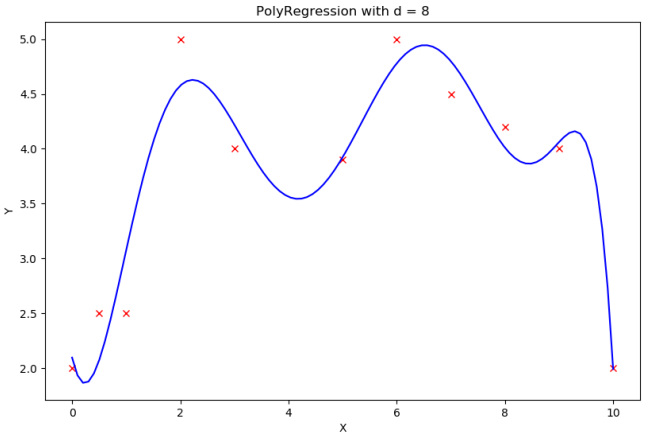
a. init(degree=1, regLambda=1E-8): constructor with arguments of d and λ

b. fit(X,Y): method to train the polynomial regression model

c. predict(X): method to use the trained polynomial regression model for prediction

d. polyfeatures(X, degree): expands the given $n \times 1$ matrix X into an $n \times d$ matrix of polynomial features of degree d. Note that the returned matrix will not include the zero-th power. Note that the polyfeatures(X, degree) function maps the original univariate data into its higher order powers. Specifically, X will be an n matrix ($X \in \mathbb{R}_n \times 1$ and this function will return the polynomial expansion of this data, a $n \times d$ matrix. Note that this function will not add in the zero-th order feature (i.e., $x_0 = 1$). You should add the x_0 feature separately, outside of this function, before training the model. By not including the x_0 column in the matrix polyfeatures(), this allows the polyfeatures function to be more general, so it could be applied to multi-variate data as well. (If it did add the x_0 feature, we'd end up with multiple columns of 1's for multivariate data.) Also, notice that the resulting features will be badly scaled if we use them in raw form. For example, with a polynomial of degree $d = 8$ and $x = 20$, the basis expansion yields $x1 = 20$ while $x_8 = 2.56 \cdot 10^{10}$ an absolutely huge difference in range. Consequently, we will need to standardize the data before solving linear regression. Standardize the data in fit() after you perform the polynomial feature expansion. You'll need to apply the same standardization transformation in predict() before you apply it to new data.

Run testpolyregunivariate.py to test your implementation, which will plot the learned function. In this case, the script fits a polynomial of degree $d = 8$ with no regularization $\lambda = 0$. From the plot, we see that the function fits the data well, but will not generalize well to new data points. Try increasing the amount of regularization, and examine the resulting effect on the function.



```
#--------------------------------------------------------------------
#  Class PolynomialRegression
#--------------------------------------------------------------------

class PolynomialRegression:
    def __init__(self, degree=1, reg_lambda=1E-8):
        """
        Constructor
        """
        self.regLambda = reg_lambda
        self.degree = degree
        self.theta = None
        self._mean = None
        self._std = None


    def _expandToDegree(self, X, degree=None):
        """Expands the given column vector to a (n,d) matrix where elements are
        powers of values x_i including the zero-th order.

        [x1, ... x_n]^T = [[1, x1, x_1^2, ... x_1d^d],
                                ...
                           [1, x_n, x_n^2, ... x_nd^d]]
        """
        if degree is None:
            degree = self.degree
        return (X[:,None]**np.arange(degree+1))[:, 0, :]


    def polyfeatures(self, X, degree):
        """
        Expands the given X into an n * d array of polynomial features of
        degree d.

        Returns:
            A n-by-d numpy array, with each row comprising of
            X, X * X, X ** 3, ... up to the dth power of X.
            Note that the returned matrix will not include the zero-th power.

        Arguments:
            X is an n-by-1 column numpy array
            degree is a positive integer
        """
        self.degree = degree
        return self._expandToDegree(X, degree)[:, 1:]


    def standardize(self, X, mean=None, std=None):
        """Returns a standardized copy of the array using the given weights.

        Standardization is performed by offsetting by mean and dividing by
        variance on a per column basis.
        """
        mean = self._mean if mean is None else mean
        std = self._std if std is None else std
        standardized = []
        for row in X:
            standardized.append((row-mean)/std)
        return np.vstack(standardized)


    def fit(self, X, y):
        """
            Trains the model
            Arguments:
                X is a n-by-1 array
                y is an n-by-1 array
            Returns:
                No return value
            Note:
                You need to apply polynomial expansion and scaling
                at first
        """
        # expand to polynomial of degree d
        X_ = self.polyfeatures(X, self.degree)

        # standardize the matrix and remember the weights
        self._mean = np.mean(X_, axis=0)
        self._std = np.std(X_, axis=0)
        X_ = self.standardize(X_)

        # add a column of oens
        X_ = np.c_[np.ones([len(X), 1]), X_]

        # construct reg matrix
        reg_matrix = self.regLambda * np.eye(self.degree+1)
        reg_matrix[0, 0] = 0

        # analytical solution (X'X + regMatrix)^-1 X' y
        self.theta = np.linalg.pinv(X_.T.dot(X_) + reg_matrix).dot(X_.T).dot(y)
        return self.theta


    def predict(self, X):
        """
        Use the trained model to predict values for each instance in X
        Arguments:
            X is a n-by-1 numpy array
        Returns:
            an n-by-1 numpy array of the predictions
        """
        X_ = self.polyfeatures(X, self.degree)
        X_ = self.standardize(X_)
        X_ = np.c_[np.ones([len(X), 1]), X_]
        return X_.dot(self.theta)


#--------------------------------------------------------------------
#  End of Class PolynomialRegression
#--------------------------------------------------------------------
```

A.5. *[10 points]* In this problem we will examine the bias-variance tradeoff through learning curves. Learning curves provide a valuable mechanism for evaluating the bias-variance tradeoff. Implement the learningCurve()function in polyreg.py to compute the learning curves for a given training/test set. The learningCurve(Xtrain,ytrain, Xtest, ytest, degree, regLambda) function should take in the training data (Xtrain,ytrain), the testing data (Xtest,ytest), and values for the polynomial degree d and regularization parameter λ.The function should return two arrays, errorTrain (the array of training errors) and errorTest(the array of testing errors). The ith index (start from 0) of each array should return the training error (or testing error) for learning with $i + 1$ training instances. Note that the 0th index actually won't matter, since we typically start displaying the learning curves with two or more instances. When computing the learning curves, you should learn on $X_{\text{train}}[0 : i]$ for $i = 1, ..., \text{numInstances(Xtrain)} + 1$,each time computing the testing error over the entire test set. There is no need to shuffle the training data, or to average the error over multiple trials - just produce the learning curves for the given training/testing sets with the instances in their given order. Recall that the error for regression problems is given by:

$$\frac{1}{n}\sum_{i=1}^{n} h_\theta(x_i) - y_i)^2$$

Once the function is written to compute the learning curves, run thetestpolyreglearningCurve.py script to plot the learning curves for various values of λ and d.

Ridge Regression on MNIST

A.6. In this problem we will implement a regularized least squares classifier for the MNIST data set. The task is to classify handwritten images of numbers between 0 to 9.You are NOT allowed to use any of the pre-built classifiers in sklearn. Feel free to use any method from numpy or scipy. Remember: if you are inverting a matrix in your code, you are probably doing something wrong (Hint: look at scipy.linalg.solve). Get the data from `https://pypi.python.org/pypi/python-mnist`. Load the data as follows:

```
from mnist import MNIST

def load_dataset():
    mndata = MNIST('./data/')
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train/255.0X_test = X_test/255.0
```

Each example has features $x_i \in R^d$ (with $d = 28 \times 28 = 784$) and label $z_j \in 0, ..., 9$. You can visualize a single example x_i with imshow after reshaping it to its original 28x28 image shape (and noting that the label z_j is accurate). We wish to learn a predictor $\hat{f}$ that takes as input a vector in R^d and outputs an index in $0, ..., 9$. We define our training and testing classification error on a predictor f as:

$$\widehat{\epsilon}_{\text{train}} = \frac{1}{N_{\text{train}}} \sum_{(x,z) \in \text{Training Set}} \mathbf{1}\{f(x) \neq z\}$$

$$\widehat{\epsilon}_{\text{train}} = \frac{1}{N_{\text{train}}} \sum_{(x,z) \in \text{Training Set}} \mathbf{1}\{f(x) \neq z\}$$

We will use one-hot encoding of the labels, i.e. of (x, z) the original label $z \in 0, ..., 9$ is mapped to the standard basis vector e_z where e_z is a vector of all zeros except for a 1 in the zth position. We adopt the notation where we have n data points in our training objective with features $x_i \in R^d$ and label one-hot encoded as $y_i \in 0, 1^k$ where in this case $k = 10$ since there are 10 digits.

a. *[10 points]* In this problem we will choose a linear classifier to minimize the regularized least squares objective:

$$\widehat{W} = \arg\min_{W \in \mathbb{R}^{d \times k}} \sum_{i=0}^{n} ||W^T x_i - y_i||_2^2 + \lambda ||W||_F^2$$

Note that $||W||_F$ corresponds to the Frobenius norm of W, i.e. $||W||_F^2 = \sum_{j=0}^{d} \sum_{i=0}^{k} W_{i,j}^2$. To classify a point x_i we will use the rule $\text{argmax}_{j=0,...9}\, e_j^T \widehat{W}^T x_i$. Note that if $W = [w_1 \dots w_k]$ then

$$\begin{aligned}\sum_{i=0}^{n} ||W^T x_i - y_i||_2^2 + \lambda ||W||_F^2 &= \sum_{j=0}^{k} \left[\sum_{i=0}^{n} (e_j^T W^T x_i - e_j^T y_i)^2 + \lambda ||W_{e_j}||^2 \right] \\ &= \sum_{j=0}^{k} \left[\sum_{i=0}^{n} (w_j^T x_i - e_j^T y_i)^2 + \lambda ||w_j||^2 \right] = \sum_{j=0}^{k} \left[||X w_j - Y e_j||^2 + \lambda ||w_j||^2 \right]\end{aligned}$$

where $X = [x_1 \dots x_n]^T \in \mathbb{R}^{n \times d}$ and $Y = [y_1 \dots y_n]^T \in \mathbb{R}^{n \times k}$. Show that:

$$\widehat{W} = (X^T X + \lambda I)^{-1} X^T Y$$

The given hints demonstrates, in simplified terms, that $\text{argmax}\, ||W^T x_i + y_i|| = \text{argmax}\, ||XW + y||$ so we apply the same to argmin and rewrite given least squares objective $\widehat{W}$ as:

$$\widehat{W} = \arg\min_{W \in \mathbb{R}^{d \times k}} ||XW - y||_2^F + \lambda ||W||_F^2$$

Taking the derivative (see HW0 A.10.b) $\nabla_x = \frac{\partial}{\partial x}^T$), with respect to W and equating with 0:

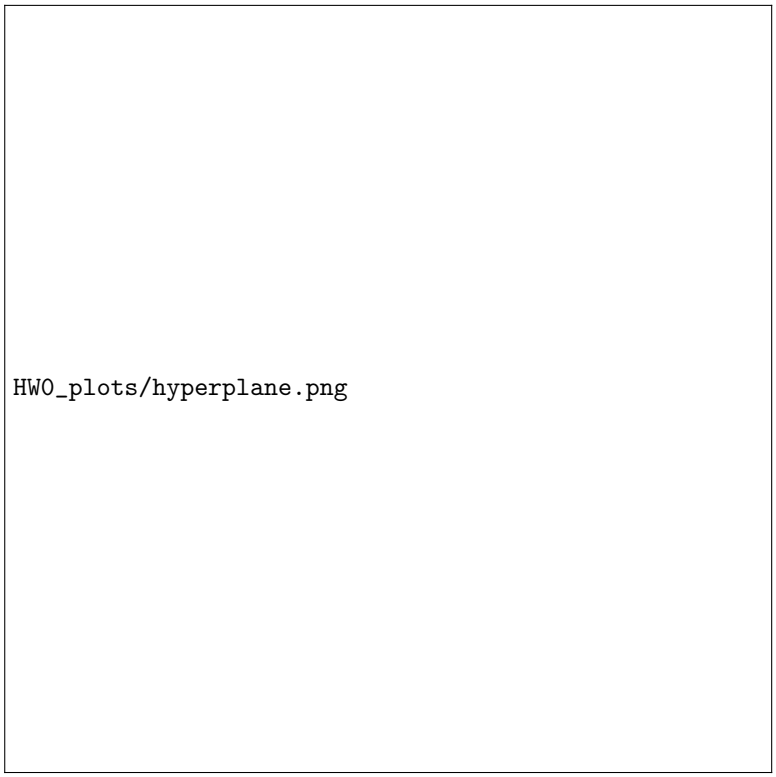
$$\begin{aligned}
\nabla_W \left(||XW||_F^2 + \lambda ||W||_F^2 \right) &= -2X^T(y - XW) + 2\lambda W = 0 \\
X^T XW - X^T y + \lambda W &= 0 \\
(X^T X + \lambda I)W &= X^T y \\
W &= (X^T X + \lambda I)^{-1} X^T y
\end{aligned}$$

```
import numpy as np
from mnist import MNIST
from scipy import linalg


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

    MNIST data are 28x28 pixel large images of letters.

    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("data/mnist_data/")

    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 one_hot(length, index):
    """Given an index and length k returns an array where all elements are zero
    except the one at index location, where the value is 1.

    Parameters
    ----------
    length : `int`
        Length of the almost-zero array.
    index : `int`
        Index at which element value is set to 1

    Returns
    -------
    arr : `np.array`
        Array of zeros except for arr[index]=1.
    """
    arr = np.zeros(length)
    arr[index] = 1
    return arr


def train(X, Y, lamb):
    """Given data, labels and regularization constant lambda solves

    $$ W = (X^T X) + \lambda I $$

    to retrieve weights of our model.

    Parameters
    ----------
    X : `np.array`
        Data to fit to
    Y : `np.array`
        Data labes, a length 10 array where index of element with value 1 marks
        the number the number respective data point x represents.
    lamb : `float`
        Regularization parameter lambda.

    Returns
    -------
    wHat : `np.array`
        Matrix of weights that minimize the linear least squares.
    """
    n, d = X.shape

    a = np.dot(X.T, X) + lamb*np.eye(d)
    b = np.dot(X.T, Y)
    wHat = linalg.solve(a, b)

    return wHat


def predict(W, data, labelDim):
    """Given weights, data and the dimension of the labels space predicts what
    label is the data most likely representing.
```

```
    Parameters
    ----------
    W : `np.array`
        Array of weights of our model.
    data : `np.array`
        Array of data to classify
    labelDim : `int`
        Label space dimension

    Returns
    -------
    classifications : `np.array`
        Array of final predicted classifications of the data.
    """
    predictions = np.dot(data, W)
    # pick out only the most probably values, i.e. the maxima
    maxPredictions = np.argmax(predictions, axis=1)
    classifications = np.array([one_hot(labelDim, y) for y in maxPredictions])
    return classifications


def calc_success_fraction(W, data, labels):
    """Given weights, data and labels predicts the labels of the data and by
    comparing them to the given labels calculates the fraction of the predicted
    classifications that were correct and wrong as a

    fracWrong = (\sum |predicted - actualLabel|) / (2*N_data)
    fracCorrect = 1 - fracWrong

    Parameters
    ----------
    W : `np.array`
        Weights of our model
    data : `np.array`
        data we want to predict labels for
    labels : `np.array`
        labels of actual class the data

    Returns
    -------
    fracCorrect : `float`
        Fraction of correctly predicted labels
    fracWrong : `float`
        Fraction of incorrectly predicted labels
    """
    n, d = data.shape
    labelDim = labels.shape[-1]

    wrong = np.sum(np.abs(predict(W, data, labelDim) - labels))
    # 2 is required because abs value will contribute double to the sum
    fracWrong = wrong/(2.0*n)
    fracCorrect = 1 - fracWrong

    return fracCorrect, fracWrong


def main(lambd=1e-4):
    """Given the dimension of label space and regularization parameter value
    trains a model on the MNIST train dataset, predicts the labels on the MNIST
    test dataset and calculates the fraction of wrongly predicted labels.

    Parameters
    ----------
    lambd : `float`
        Regularization parameter (lamda)

    Returns
    -------
    trainErr : `float`
        Training error, fraction of incorrectly labeled train data
    testErr : `float`
        Test error, fraction of incorrectly labeled test data
    """
    xTrain, trainLabels, xTest, testLabels = load_mnist_dataset()
    n, d = xTrain.shape
    labelDim = trainLabels.max() + 1

    yTrain = np.array([one_hot(labelDim, y) for y in trainLabels])
    yTest  = np.array([one_hot(labelDim, y) for y in testLabels])

    wHat = train(xTrain, yTrain, lambd)

    trainErr = calc_success_fraction(wHat, xTrain, yTrain)[-1]
    testErr  = calc_success_fraction(wHat, xTest,  yTest)[-1]
    print(f"Train error: {trainErr}")
    print(f"Test error: {testErr}")

    return trainErr, testErr


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

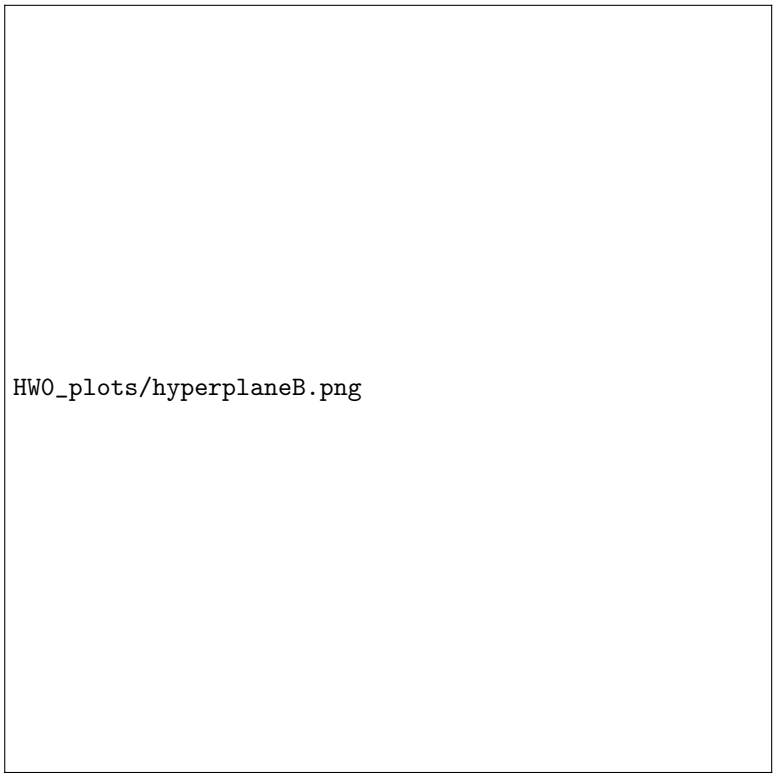
A.9 (Hyperplanes) Assume w is an n-dimensional vector and b is a scalar. A hyperplane in $\mathbb{R}^n$ is the set $\{x : x \in \mathbb{R}^n, \text{ s.t. } w^T x + b = 0\}$.

a. *[1 points]* ($n = 2$ example) Draw the hyperplane for $w = [-1, 2]^T$, $b = 2$? Label your axes.
Effectively this gives us an equation of a line $-x_1 + 2x_2 + 2 = 0$ or $x_1 = 2x_2 + 2$. Plot line using following Python code:




```
import matplotlib.pyplot as plt
import numpy as np
x2 = np.arange(-10, 10, 1)
x1 = 2*x2+2
plt.plot(x1, x2)
plt.xlabel("x2")
plt.ylabel("x1")
plt.grid()
```

b. *[1 points]* ($n = 3$ example) Draw the hyperplane for $w = [1, 1, 1]^T$, $b = 0$? Label your axes.
Following the same principles above $x + y + z = 0 \rightarrow z = -x - y$ we have:




```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import numpy as np

x2 = np.arange(-10, 10, 1)
x3 = np.arange(-10, 10, 1)
X2, X3 = np.meshgrid(x2, x3)
X1 = -X2 -X3

fg = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X1, X2, X3, 150)
ax.set_zlabel("x3")
ax.set_xlabel("x2")
ax.set_ylabel("x1")
plt.grid()
```

c. *[2 points]* Given some $x_0 \in \mathbb{R}^n$, find the *squared distance* to the hyperplane defined by $w^T x + b = 0$. In other words, solve the following optimization problem:

$$\min_x \|x_0 - x\|^2$$
$$\text{s.t. } w^T x + b = 0$$

(Hint: if $\widetilde{x}_0$ is the minimizer of the above problem, note that $\|x_0 - \widetilde{x}_0\| = |\frac{w^T(x_0 - \widetilde{x}_0)}{\|w\|}|$. What is $w^T \widetilde{x}_0$?) The following solution was hinted to me by Conor Sayers. Conors idea was that we are given a solution for non-squared distance and have enough information in the hint of the problem to "reverse engineer" the squared distance solution. If $\widetilde{x}_0$ is the vector that minimizes the problem we can write $w^T \widetilde{x}_0 = -b$ and then we can write the squared distance as:

$$\begin{aligned} |x_0 - \widetilde{x}_0|^2 &= \left[\frac{w^T(x_0 - \widetilde{x}_0)}{|w|}\right]^2 \\ &= \left[\frac{w^T x_0 - w^T \widetilde{x}_0}{|w|}\right]^2 \\ &= \left[\frac{w^T x_0 + b}{|w|}\right]^2 \end{aligned}$$

The original attempt I have left below to, hopefully, show that I hadn't dilly-dallied around waiting for a solution to be found by someone else and thus, hopefully, avoid accusations of plagiarism:
We are looking to minimize the following function:

$$\begin{aligned} f(\vec{x}) &= \omega^T ||\vec{x} - \vec{x}_0||^2 + b = 0 \\ &= \omega^T ||\vec{x}^T\vec{x} - 2\vec{x}\cdot\vec{x}_0 + \vec{x}_0^T\vec{x}_0|| + b \\ &= \omega\vec{x}^T\vec{x} - 2\omega^T||\vec{x}\cdot\vec{x}_0|| + \omega^T\vec{x}_0^T\vec{x}_0 + b = 0 \end{aligned}$$

Note that values $\vec{x}^T\vec{x}$ and $\vec{x}_0^T\vec{x}_0$ are just scalars. Minimization can be performed via the method of Lagrange multipliers of the above equation given the constraint that $\vec{x}$ lies in the plane $\Phi = \omega^T\vec{x} + b = 0$:

$$\begin{aligned} \frac{\partial f(\vec{x})}{\partial x_i} - \lambda\frac{\partial\Phi(\vec{x})}{\partial x_i} &= 0 \\ -2\omega^T\frac{\partial||\vec{x}\cdot\vec{x}_0||}{\partial x_i} - \lambda\omega^T\frac{\partial\vec{x}}{\partial x_i} &= 0 \\ -2\omega^T\frac{\partial\vec{x}}{\partial x_i}\cdot\vec{x}_0 - \lambda\omega^T\frac{\partial\vec{x}}{\partial x_i} &= 0 \end{aligned}$$

But of course now I run into a problem where I can not easily work out the system of equations.

A.10 For possibly non-symmetric $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n\times n}$ and $c \in \mathbb{R}$, let $f(x, y) = x^T \boldsymbol{A} x + y^T \boldsymbol{B} x + c$. Define $\nabla_z f(x, y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial z_1} & \frac{\partial f(x,y)}{\partial z_2} & \dots & \frac{\partial f(x,y)}{\partial z_n} \end{bmatrix}^T$.

a. *[2 points]* Explicitly write out the function $f(x, y)$ in terms of the components $A_{i,j}$ and $B_{i,j}$ using appropriate summations over the indices.
Not sure what is meant by this but this is how the function would look like written out "explicitly":

$$f(x,y) = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \dots & A_{nn} \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \begin{bmatrix} B_{11} & \dots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \dots & B_{nn} \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix} + c$$

Writing the above in terms of summations over indices we keep in mind the definition of matrix multiplication: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Applied to Ax for example we have $Ax = \sum_{i=1}^{n} A_{1,i} x_i$. We write the following:

$$f(x,y) = \sum_{i=1}^{n} x_i \sum_{j=1}^{n} B_{ij} x_j + \sum_{i=1}^{n} y_i \sum_{j=1}^{n} B_{ij} x_j + c$$

b. *[2 points]* What is $\nabla_x f(x, y)$ in terms of the summations over indices *and* vector notation?
In this context ∇_x represents the derivation operator with respect to elements of vector x. In summation form, if coordinates are orthogonal, it's defined as: $\nabla f = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \frac{1}{h_i} \hat{e}_i$ where h are the Lame's coefficients (important for coordinate systems except for Cartesian where they're equal to 1). Effectively ∇_x turns a column vector to a row vector so in vector notation we can write $\nabla_x = \frac{\partial}{\partial x}^T$. Applied to $f(x, y)$:

$$\nabla_x f(x,y) = \nabla_x (x^T A x) + \nabla_x (y^T B x) = \frac{\partial}{\partial x_i} \sum_{j=1}^{n} \sum_{k=1}^{n} x_j A_{jk} x_k + \frac{\partial}{\partial x_i} \sum_{j=1}^{n} \sum_{k=1}^{n} y_j B_{jk} x_k$$

where $i \in \{1, ..., n\}$. Starting with the first term[1]:

$$\begin{aligned} \frac{\partial}{\partial x_i} \sum_{j=1}^{n} \sum_{k=1}^{n} x_j A_{jk} x_k &= \sum_{j=1}^{n} \sum_{k=1}^{n} \frac{\partial x_j}{\partial x_i} A_{jk} x_k + \sum_{j=1}^{n} \sum_{k=1}^{n} x_j A_{jk} \frac{\partial x_k}{\partial x_i} \\ &= \sum_{k=1}^{n} A_{ik} x_k + \sum_{j=1}^{n} x_j A_{ji} = \sum_{k=1}^{n} A_{ik} x_k + \sum_{j=1}^{n} A_{ji} x_j \end{aligned}$$

In the first term the free index i iterates over the columns of A, and in the second term over the rows of A. Meanwhile, there is really no difference between indices k and j with respect to vector elements of x except to indicate the correct element of A within the sums themselves, additionally we know in general that matrix multiplication is distributive with respect to addition, so we can write the two individual sums over a single sum. :

$$\sum_{k=1}^{n} A_{ik} x_k + \sum_{j=1}^{n} A_{ji} x_j = \sum_{j=1}^{n} (A_{ij} + A_{ji}) x_j$$

For a fixed i, we recognize the retrieved expression as the i-th element of a matrix expression $[(A + A^T)x]_i$ so finally we can write the result in vector form:

$$\nabla_x (x^T A x) = (A + A^T) x$$

For the second term we, similarly, write:

$$\begin{aligned} \frac{\partial}{\partial x_i} \sum_{j=1}^{n} \sum_{k=1}^{n} y_j B_{jk} x_k &= \sum_{j=1}^{n} \sum_{k=1}^{n} \frac{\partial y_j}{\partial x_i} B_{jk} x_k + \sum_{j=1}^{n} \sum_{k=1}^{n} y_j B_{jk} \frac{\partial x_k}{\partial x_i} \\ &= \sum_{j=1}^{n} y_j B_{ji} = \left[B^T y \right]_i \end{aligned}$$

[1] Note that the use of product rule is optional as the same could be achieved by carefully working out $\partial_i (x_j x_k)$ keeping track of the indices. It's just more obvious as a product rule.

Which we recognize as the vector expression $\nabla_x(y^T Bx) = B^T y$. Put together we have that

$$\nabla_x f(x, y) = (A + A^T)x + B^T y$$

c. *[2 points]* What is $\nabla_y f(x, y)$ in terms of the summations over indices *and* vector notation?
Similarly to the previous problem except that we only have to observe a single term that has a functional dependence on y:

$$\begin{aligned}\frac{\partial}{\partial y_i}\sum_{j=1}^{n}\sum_{k=1}^{n} y_j B_{jk} x_k &= \sum_{j=1}^{n}\sum_{k=1}^{n}\frac{\partial y_j}{\partial y_i} B_{jk} x_k + \sum_{j=1}^{n}\sum_{k=1}^{n} y_j B_{jk}\frac{\partial x_k}{\partial y_i} \\ &= \sum_{j=1}^{n} B_{ik} x_k = [Bx]_i\end{aligned}$$

From which we recognize the vector expression $\nabla_y(y^T Bx) = Bx$. Since the remaining terms in the expression will evaluate to zero, as there is no dependence on y, we can immediately write:

$$\nabla_y f(x, y) = Bx$$

Programming

A.11 For the A, b, c as defined in Problem 8, use NumPy to compute (take a screen shot of your answer):

a. *[2 points]* What is A^{-1}?

b. *[1 points]* What is $A^{-1}b$? What is Ac?

HW0_plots/numpy.png

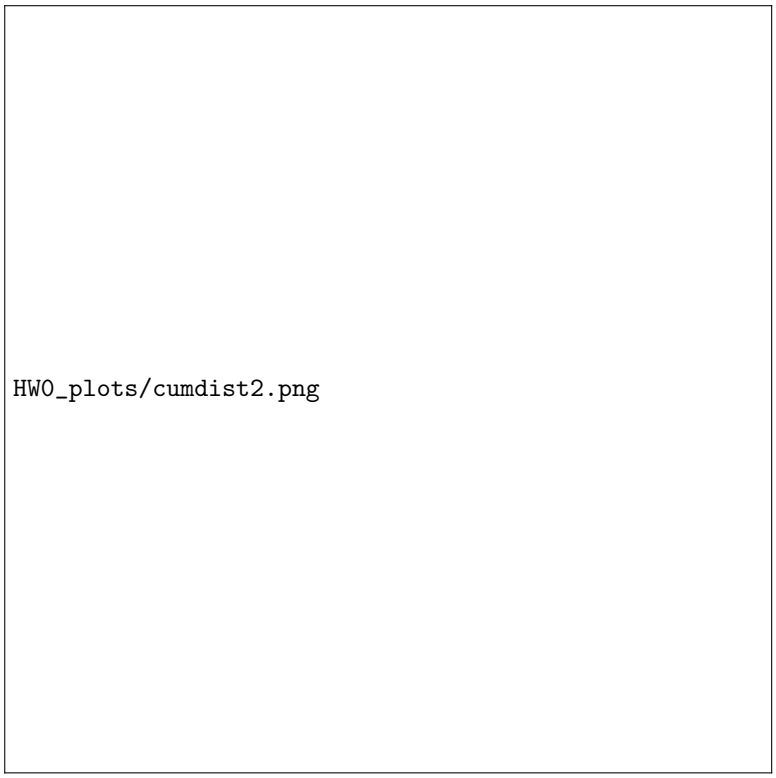
A.12 *[4 points]* Two random variables X and Y have equal distributions if their CDFs, F_X and F_Y, respectively, are equal, i.e. for all x, $|F_X(x) - F_Y(x)| = 0$. The central limit theorem says that the sum of k independent, zero-mean, variance-$1/k$ random variables converges to a (standard) Normal distribution as k goes off to infinity. We will study this phenomenon empirically (you will use the Python packages Numpy and Matplotlib). Define $Y^{(k)} = \frac{1}{\sqrt{k}} \sum_{i=1}^{k} B_i$ where each B_i is equal to -1 and 1 with equal probability. From your solution to problem 5, we know that $\frac{1}{\sqrt{k}} B_i$ is zero-mean and has variance $1/k$.

a. For $i = 1, \ldots, n$ let $Z_i \sim \mathcal{N}(0, 1)$. If $F(x)$ is the true CDF from which each Z_i is drawn (i.e., Gaussian) and $\widehat{F}_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{Z_i \leq x\}$, use the answer to problem 1.5 above to choose$n$ large enough such that, for all $x \in \mathbb{R}$, $\sqrt{\mathbb{E}[(\widehat{F}_n(x) - F(x))^2]} \leq 0.0025$, and plot $\widehat{F}_n(x)$ from -3 to 3.
(Hint: use `Z=numpy.random.randn(n)` to generate the random variables, and `import matplotlib.pyplot as plt`; `plt.step(sorted(Z), np.arange(1,n+1)/float(n))` to plot).
From problem A6 we have $\mathrm{Var}\widehat{F}_n(x) = (4n)^{-1}$ so plugging in the value 0.0025 given in the problem we have $n = (0.0025 * 4)^{-1} = 100$ so just to be on the very safe side lets sample several tens of thousand times.

HW0_plots/cumdist1.png

```
import matplotlib.pyplot as plt
import numpy as np

def Fn(Z, x=1.0):
    """Calculates the empirical estimate of the
    CDF F(x) by counting the number of occurences
    of random variable Z up to the limit x.

    Parameters
    ----------
    Z : `array`
        An array of samples of random variable Z
    x : `float`
        Limit to which the empirical estimate is calculated.
    """
    return np.sum(Z<=x)/len(Z)

n = 20000
Z = np.random.randn(n)
x = np.arange(-3, 3, 0.01)
fn = [Fn(Z, _x) for _x in x]

plt.step(x, fn)
plt.xlim(-3, 3)
plt.xlabel("Observations")
plt.ylabel("Probability")
plt.show()
```

b. For each $k \in \{1, 8, 64, 512\}$ generate n independent copies $Y^{(k)}$ and plot their empirical CDF on the same plot as part a.
(Hint: `np.sum(np.sign(np.random.randn(n, k))*np.sqrt(1./k), axis=1)` generates n of the $Y^{(k)}$ random variables.)




```
import matplotlib.pyplot as plt
import numpy as np

def Yk(n, k=1):
    """Returns and array of samples of function:

        Y^{(k)} = 1/sqrt(k) \sum_1^k B_i

    where B_i = +1 or +1 with equal probablility.

    Parameters
    ----------
    n : `int`
        Number of samples of Y^(k).
    k : `int`
        Upper bound of the Y^(k) sum
    """
    B = np.sign(np.random.randn(n, k))
    return np.sum(np.sqrt(1.0/k)*B, axis=1)


n = 20000
for k in [1, 8, 64, 512]:
    Y = Yk(n, k)
    plt.step(sorted(Y), np.arange(1, n + 1) / float(n), label='{0}'.format(k))

gaus = np.random.normal(size=n)
plt.step(sorted(gaus), np.arange(1, n + 1) / float(n), label='Gaussian')

plt.legend()
plt.xlim(-3, 3)
plt.xlabel("Observations")
plt.ylabel("Probability")
plt.show()
```