Homework #1 A

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

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

Conceptual Questions *[10 points]*

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

a. *[2 points]* Suppose that your estimated model for predicting house prices has a large positive weight on 'number of bathrooms'. Does it implies that if we remove the feature "number of bathrooms" and refit the model, the new predictions will be strictly worse than before? Why?
If number of bathrooms is independent feature our predictions would be worse. But as we discussed in class, often it is hte case that number of bathrooms is really not an independent feature in the feature set, but instead the weights are split between several different factors that directly or indirectly measure the number of bathrooms.

b. *[2 points]* Compared to L2 norm penalty, explain why a L1 norm penalty is more likely to result in a larger number of 0s in the weight vector or not?
In class we said L1 penalty leads to sparse outputs.

c. *[2 points]* In at most one sentence each, state one possible upside and one possible downside of using the following regularizer: $\sum_i |w_i|^{0.5}$)

d. *[1 points]* True or False: If the step-size for gradient descent is too large, it may not converge.
True. When jumps occur with a step-size much greater than the characteristic step length of a function we can end up constantly jumping to higher values and further away from the minimum.

e. *[2 points]* In your own words, describe why SGD works.
Gradient descent effectively works by starting from a random point and then calculating gradient at that point with respect to all features in the dataset in order to pick the direction of the next point it will step to. Computing true gradients of the sum of squared residuals with respect to features is very costly with datasets containing a lot of points for even moderate number of features. Stochastic gradient descent computes the gradient using a randomly selected point, or a set of points, instead of computing the true gradient using all points. This obviously works for relatively monotonic functions since what we have effectively done was approximating the true function with a line or a hyperplane.

f. *[2 points]* In at most one sentence each, state one possible advantage of SGD (stochastic gradient descent)over GD (gradient descent) and one possible disadvantage of SGD relative to GD.
SGD is much less computationally demanding, but might take a longer time to converge to minimum value than GD might. GD on the other hand virtually guarantees that fewer steps will need to be taken to converge to minimum compared to SGD.

Convexity and Norms *[10 points]*

A.1. A norm $||\cdot||$ over R^n is defined by the properties:

(a) non-negative: $||x|| \geq 0 \forall x \in \mathbb{R}^n \iff x = 0$,

(b) absolute scalability: $||ax|| = |a|||x|| \forall a \in \mathbb{R}$ and $x \in \mathbb{R}^n$,

(c) triangle inequality: $||x + y|| \leq ||x|| + ||y|| \forall x, y \in \mathbb{R}^n$

a. Show that $f(x) = (\sum_{i=1}^{n} |x_i|)$ is a norm. (Hint: begin by showing that $|a + b| \leq |a| + |b| \forall a, b \in \mathbb{R}$.
Trick is to notice that norm is defined over a field we are very familiar with and which rules we know. Since $x_i \in \mathbb{R} \rightarrow |x_i| \geq 0$ by definition of absolute value of a number we can write:

$$f(\vec{x}) = ||\vec{x}|| = \sum |x_i| \geq 0$$

We show absolute homogeneity using same trick:

$$f(a\vec{x}) = ||a\vec{x}|| = \sum |ax_i| = \sum |a||x_i| = |a| \sum |x_i| = |a|||\vec{x}|| = |a|f(x)$$

where first we applied the norm we are testing, wrote out its definition, found ourselves operating on elements of $\mathbb{R}$ where associativity and multiplicativity applies, rewrote so it suits our purpose, and then walked back up the chain. Triangle inequality is likely the most interesting property to prove. Question requires us to prove the triangle inequality in $\mathbb{R}$ first. For any $a, b \in \mathbb{R}$ from definition of absolute value it follows:

$$-|a| \leq a \leq |a|$$
$$-|b| \leq b \leq |b|$$

Summing the first two rows we have

$$-(|a| + |b|) \leq a + b \leq |a| + |b|$$

Since $|c| \leq d \rightarrow -d \leq c \leq d$, we identify $c := a + b$ and $d := |a| + |b|$ we rewrite the line above as:

$$|a + b| \leq |a| + |b|$$

This proof is effectively a clearer more verbose version of Wikipedia proof. It's hard to find yet another way to prove triangle inequality. Applying the triangle relativity in $\mathbb{R}$ to our problem:

$$\begin{aligned} f(\vec{x} + \vec{y}) = ||\vec{x} + \vec{y}|| &= \sum |x_i + y_i| \\ &\leq \sum (|x_i| + |y_i|) \\ &= \sum |x_i| + \sum |y_i| \\ &= ||\vec{x}|| + ||\vec{y}|| = f(\vec{x}) + f(\vec{y}) \\ \rightarrow f(\vec{x} + \vec{y}) &\leq f(\vec{x}) + f(\vec{y}) \end{aligned}$$

b. *[2 points]* Show that $g(x) = (\sum_{i=1}^{n} |x_i|^{1/2})^2$ is not a norm. (Hint: it suffices to find two points in $n = 2$ dimensions such that the triangle inequality does not hold.)
Since:

$$(\sqrt{|x_i|} + \sqrt{|y_i|})^2 = |x_i| + |y_i| + 2\sqrt{|x_i||y_i|} \geq |x_i| + |y_i|$$
$$(\sqrt{|x_i|} + \sqrt{|y_i|})^2 \geq |x_i| + |y_i|$$
$$\sqrt{|x_i|} + \sqrt{|y_i|} \geq \sqrt{|x_i| + |y_i|}$$

Applying the norm definition and the inequality shown above:

$$\begin{aligned} g(\vec{x} + \vec{y}) &= \sum \sqrt{|x_i + y_i|}^2 \\ &\leq \sum (\sqrt{|x_i| + |y_i|})^2 \\ &\leq \sum (\sqrt{|x_i|} + \sqrt{|y_i|})^2 \\ g(\vec{x} + \vec{y}) &\leq \sum \left(|x_i| + |y_i| + 2\sqrt{|x_i||y_i|} \right) \\ g(\vec{x} + \vec{y}) &\leq g(\vec{x}) + g(\vec{y}) + 2 \sum \sqrt{|x_i||y_i|} \end{aligned}$$

Therefore it's not clear that the triangle inequality holds.

Context: norms are often used in regularization to encourage specific behaviors of solutions. If we define $||x||_p := (\sum_i^n = \mathbf{1}|x_i|^p)^{1/p}$ then one can show that $||x||^p$ is a norm for all $p \geq 1$. The important cases of $p = 2$ and $p = 1$ correspond to the penalty for ridge regression and the lasso, respectively.

A 2.*[3 points]* A set $A \subseteq \mathbb{R}^n$ is convex if $\lambda x + (1-\lambda)y \in A \forall x, y \in A$ and $\lambda \in [0,1]$. For each of the grey-shaded sets (I-III), state whether each one is convex, or state why it is not convex using any of the points a,b,c,d in your answer.
I is not convex because we exit the image if we connect b and c.
II is convex, there are no combinations of points that would exit the image.
III is not convex because it is possible to connect points d and a in a way that exists the image.

A 3.*[4 points]* We say a function $f : \mathbb{R}^d \to \mathbb{R}$ is convex on a set $A \iff (\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \forall x, y \in A$ and $\lambda \in [0, 1]$. For each of the grey-colored functions (I-III), state whether each one is convex on the given interval or state why not with a counter example using any of the points a,b,c,d in your answer

a. Function in panel I on [a,c]: Function is convex as there are no points that exit the graph.

b. Function in panel II on [a,c]: Function is not convex as it's possible to connect b and c such that we exit the graph.

c. Function in panel III on [a,d]: Connecting f(a) to f(d) exists the graph - not convex.

d. Function in panel III on [c,d]: Connecting f(c) to f(d) does not exit the graph, - convex in that interval.

Lasso *[45 points]*

Given $\lambda > 0$ and data $(x_1, y_1), ..., (x_n, y_n)$, the Lasso is the problem of solving

$$\arg \min_{w \in \mathbb{R}^d, b \in \mathbb{R}^n} \sum_{i=1}^{n} (x_i^T w + b - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j|$$

λ is a regularization tuning parameter. For the programming part of this homework, you are required to implement the coordinate descent method of Algorithm 1 that can solve the Lasso problem. You may use common computing packages (such as NumPy or SciPy), but do not use an existing Lasso solver (e.g., of scikit-learn).

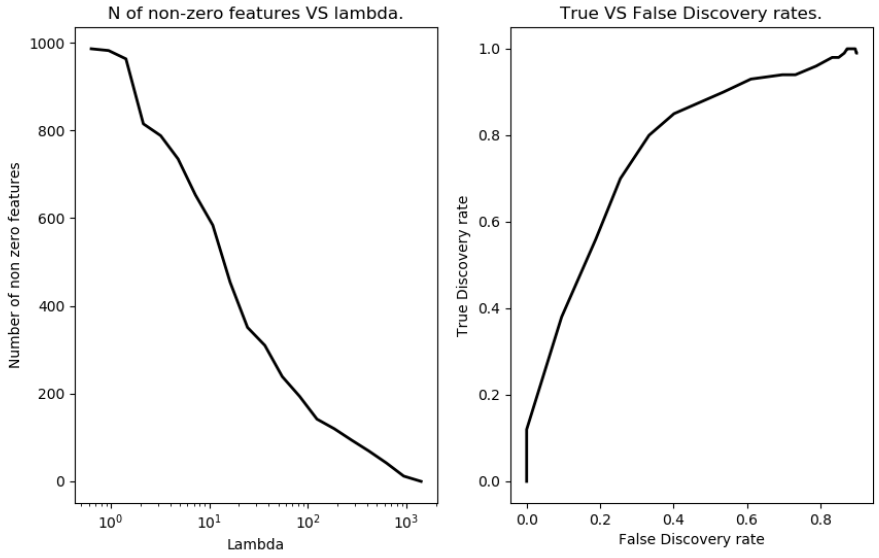
A4. We will first try out your solver with some synthetic data. A benefit of the Lasso is that if we believe many features are irrelevant for predicting y, the Lasso can be used to enforce a sparse solution, effectively differentiating between the relevant and irrelevant features. Suppose that $x \in \mathbb{R}^d, y \in \mathbb{R}, k < d$, and pairs of data (x_i, y_i) for $i = 1, \dots n$ are generated independently according to the model $y_i = w^T x_i + \epsilon_i$ where

$$h(z) = \begin{cases} j/k & \text{if } j \in \{1, \dots, k\} \\ 0 & \text{otherwise} \end{cases}$$

where $\epsilon_i \approx \mathcal{N}(0, \sigma^2)$ is some Gaussian noise (in the model above $b = 0$). Note that since $k < d$, the features $k + 1$ through d are unnecessary (and potentially even harmful) for predicting y. With this model in mind, let $n = 500, d = 1000, k = 100$, and $\sigma = 1$. Generate some data by choosing $x_i \in \mathbb{R}^d$, where each component is drawn from a $\mathcal{N}(0, 1)$ distribution and y_i generated as specified above.

a. *[10 points]* With your synthetic data, solve multiple Lasso problems on a regularization path, starting at $\lambda_{\max}$ where 0 features are selected and decreasing λ by a constant ratio (e.g., 1.5) until nearly all the features are chosen. In plot 1, plot the number of non-zeros as a function of λ on the x-axis (Tip: use log scale).

b. *[10 points]* For each value of λ tried, record values for false discovery rate (FDR) (number of incorrect non zeros in $\widehat{w}$/total number of non zeros in $\widehat{w}$) and true positive rate (TPR) (number of correct non zeros in $\widehat{w}/k$). In plot 2, plot these values with the x-axis as FDR, and the y-axis as TPR and note that in an ideal situation we would have an (FDR,TPR) pair in the upper left corner, but that can always trivially achieve $(0, 0)$ and $(d - kd, 1)$.
For both a and b parts of the problem we have the following graph:



c. *[5 points]* Comment on the effect of λ in these two plots.
No features are selected in the first step, i.e. close to $\lambda_{\max}$. Successive iterations relax regularization penalties so more and more features are selected. Simultaneously TPR increases very rapidly initially and then tapers off the rapid growth. This occurs because we quickly learn the most important features after which adding more features just doesn't add that much more valuable information to the model. Simultaneously our FDR skyrockets because the additional features over-specify our model. Eventually TPR jumps to nearly one as, I assume, we just fit all the points.

Conclusions provided courtesy of my brain, figures provided courtesy of the following code:

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


def coordinate_descent(x, y, lambd, tolerance=0.001, initW=None, convergeFast=True):
    """Preforms coordinate coordinate descent Lasso algorithm on the given data.

    Parameters
    ----------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.
    lambd: `float`
        Regularization parameter.
    tolerance: `float`, optional
        Convergence is achieved when the convergence criterion is smaller than
        tolerance.
    convergeFast: `bool`, optional
        When True, the convergence is calculated as the difference between
        maximum of new weights and maximal value of old weights; implying that
        it's likelythe two actually point to the same feature. This is not
        optimal, bu is much faster than calculating absolute value of minimal
        difference of the old and the new weights, as it's possible to avoid
        storing the old weights. True by default.
    initW: `np.array`
        Initial weights, a vector of d feature weights.

    Returns
    -------
    w: `np.array`
        New feature weights estimates.
    """
    n, d = x.shape

    if initW is None:
        w = np.zeros(d)
    else:
        w = initW

    # precalculate values used in the loop in advance
    squaredX =  2.0 * x**2

    # ensure convergence is not met on first loop
    convergeCriterion = tolerance + 1
    while convergeCriterion > tolerance:
        if convergeFast:
            # not optimal test, but fast
            oldMax = w.max()
            deltas = []
        else:
            oldW = w.copy()

        # Algorithm 1 implementation
        b = np.mean(y - np.dot(x, w))
        for k in range(d):
            xk = x[:, k]
            ak = squaredX[:,k].sum()

            # ck sum must ignore k-th dimension so we set it to zero and use
            # dot product. This matches the definition of w too, so we can
            # leave it wth zero value unless smaller than -lambda or bigger
            # than lambda anyhow.
            w[k] = 0
            delta = 0
            ck = 2.0 * np.dot(xk, y - (b + np.dot(x, w)))

            if ck < -lambd:
                delta = (ck + lambd) / ak
                w[k] = delta
            elif ck > lambd:
                delta = (ck - lambd) / ak
                w[k] = delta

            if convergeFast:
                deltas.append(delta)
        if convergeFast:
            # Find maximum difference between iterations
            convergeCriterion = abs(oldMax-max(deltas))
        else:
            convergeCriterion = np.max(np.abs(oldW-w))

    return w


def generate_data(n, d, k, sigma):
    """Generates i.i.d. samples of the model:
        y_i = w^T x_i + eps
    where
```

```
        w_j = j/k if j in {1,...,k}
        w_j = 0 otherwise
    and epsilon is random Gussian noise with the given sigma and X are also
    drawn from a Normal distribution with sigma 1.

    Parameters
    ----------
    n : `int`
        Number of samples drawn at random from the model.
    d : `int`
        Dimensionality of the feature space.
    k : `int`
        Cutoff point after which elements of w are zero.

    Returns
    -------
    x : `np.array`
        n-by-d sized array of data.
    y : `np.array`
        Vector of n model values.
    """
    # gaussian noise and data
    eps = np.random.normal(0, sigma**2, size=n)
    x = np.random.normal(size=(n, d))

    # weights
    w = np.arange(1, d + 1) / k
    w[k:] = 0

    # labels
    y = np.dot(x, w) + eps
    return x, y


def plot(ax, x, y, label="", xlabel="", ylabel="", title="", xlog=True, lc='black', lw=2):
    """Plots a line on given axis.

    Parameters
    ----------
    ax: `matplotlib.pyplot.Axes`
       Axis to plot on.
    x: `np.array`
       X axis values
    y: `np.array`
       Y axis values
    label: `str`, optional
       Line label
    xlabel: `str`, optional
       X axis label
    ylabel: `str`, optional
       Y axis label
    title: `str`, optional
       Axis title
    xlog : `bool`, optional
       X axis scaling will be logarithmic
    lc : `str`, optional
       Line color
    lw: `int` or `float`, optional
       Line width

    Returns
    ------
    ax: `matplotlib.pyplot.Axes`
        Modified axis.
    """
    ax.set_title(title)
    ax.plot(x, y, label=label, color=lc, linewidth=lw)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    if xlog:
        ax.set_xscale('log')
    return ax


def lambda_max(x, y):
    """The smallest value of regularization parameter lambda for which the
    w is entirely zero.

    Parameters
    ----------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.

    Returns
    -------
    lambda: `float`
        Smallest lambda for which w is entirely zero.
    """
    return np.max(2 * np.abs(np.dot((y - np.mean(y)), x)))


def A4_setup(n=500, d=1000, k=100, sigma=1):
    """Creates data as instructed by A4 problem, calculates the smallest value
    of regularization parameter for which w is zero and returns data parameters,
    data and calculated lambda.

    Parameters
    ----------
    n : `int`, optional
        Number of samples drawn at random from the model. Default: 500
    d : `int`, optional
        Dimensionality of the feature space. Default: 1000
    k : `int`, optional
        Cutoff point after which elements of w are zero. Default: 100
    sigma: `float`, optional
        STD of the noise Gaussian distribution that is added to the data (see
        generate_data). Default: 1.
```

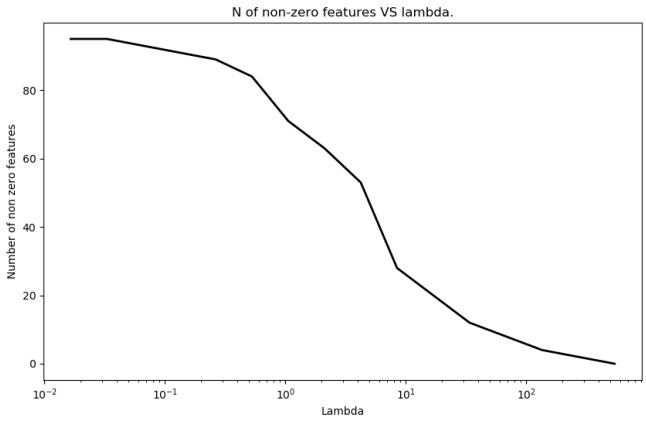
```
    Returns
    -------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.
    maxLambda: `float`
        Lambda for which w is zero everywhere.
    params: `dict`
        Dictionary of parameters used to create the data (n, d, k and sigma).
    """
    params = {'n': n, 'd': d, "k": k, "sigma": sigma}
    x, y = generate_data(n, d, k, sigma)
    maxLambda = lambda_max(x, y)
    return x, y, maxLambda, params


def A4(nIter=20, tolerance=0.001):
    """Sets the data up as instructed by problem A4 and runs coordinate
    descent Lasso algorithm nIter times, each time decreasing regularization
    parameter lambda by a factor of 1.5.
    Plots the number of non-zero-features against used lambda and false vs
    true discovery rates.

    Displays plots.

    Parameters
    ----------
    nIter: `int`, optional
        Number of different regularization parameter iterations to run. Default
        is 20.
    tolerance: `float`, optional
        Coordinate descent tolerance, sets convergence criteria (see coordinate_descent).
        Default: 0.001.
    """
    x, y, lambd, params = A4_setup()
    k = params['k']

    lambdas, numNonZeros, fdrs, tprs = [], [], [], []
    w = np.zeros(params['d'])
    for i in range(nIter):
        w = coordinate_descent(x, y, lambd, tolerance)

        nonZeros = np.count_nonzero(w)
        correctNonZeros = np.count_nonzero(w[:k])
        incorrectNonZeros = np.count_nonzero(w[k+1:])

        try:
            fdrs.append(incorrectNonZeros/nonZeros)
        except ZeroDivisionError:
            fdrs.append(0)
        tprs.append(correctNonZeros/k)

        lambdas.append(lambd)
        numNonZeros.append(nonZeros)

        lambd /= 1.5

    fig, axes = plt.subplots(1, 2, figsize=(10, 6))
    plot(axes[0], lambdas, numNonZeros, xlabel="Lambda",
         ylabel="Number of non zero features",
         title="N of non-zero features VS lambda.")
    plot(axes[1], fdrs, tprs, xlabel="False Discovery rate",
         ylabel="True Discovery rate", title="True VS False Discovery rates.",
         xlog=False)
    plt.show()


if __name__ == "__main__":
    A4()
```

A5. Now we put the Lasso to work on some real data. Download the training data set "crime-train.txt" and the test data set "crime-test.txt" from the website under Homework 2. Store your data in your working directory and read in the files with:

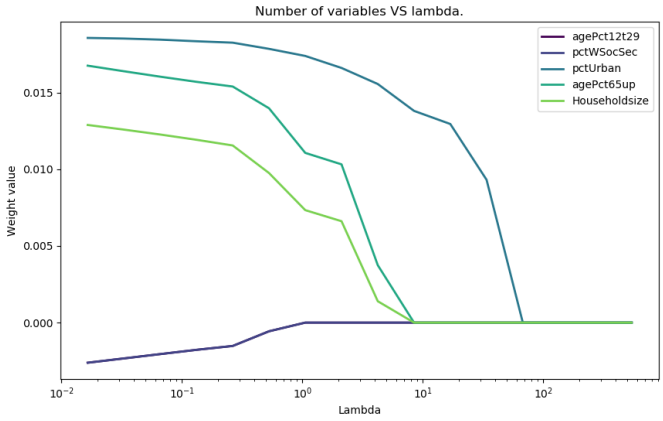
```
import pandas as pd
df_train = pd.read_table("crime-train.txt")
df_test = pd.read_table("crime-test.txt")
```

The data consist of local crime statistics for 1,994 US communities. The response y is the crime rate. The name of the response variable isViolentCrimesPerPop, and it is held in the first column of df_train and df_test. There are 95 features. These features include possibly relevant variables such as the size of the police force or the percentage of children that graduate high school. The data have been split for you into a training and test set with 1,595 and 399 entries, respectively.
We'd like to use this training set to fit a model which can predict the crime rate in new communities and evaluate model performance on the test set. As there are a considerable number of input variables, over fitting is a serious issue. In order to avoid this, use the coordinate descent LASSO algorithm you just implemented in the previous problem. Begin by running the LASSO solver with $\lambda = \lambda_{\max}$ defined above. For the initial weights, just use 0. Then, cut λ down by a factor of 2 and run again, but this time pass in the values of $\widehat{w}$ from your $\lambda = \lambda_{\max}$ solution as your initial weights. This is faster than initializing with 0 weights each time. Continue the process of cutting λ by a factor of 2 until the smallest value of λ is less than 0.01. For all plots use a log-scale for the λd dimension.

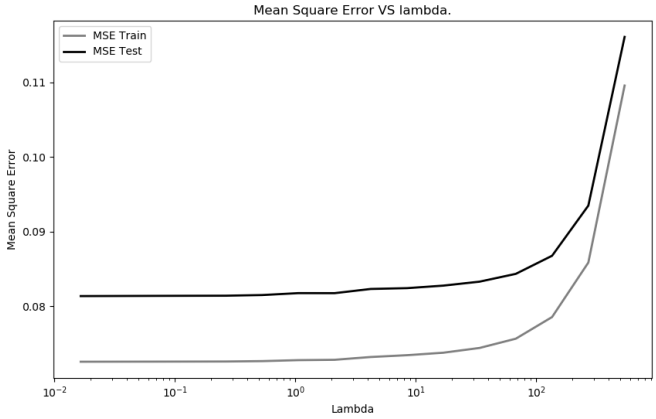
a. *[4 points]* Plot the number of nonzeros of each solution versus λ



b. *[4 points]* Plot the regularization paths (in one plot) for the coefficients for input variables agePct12t29, pctWSocSec, pctUrban, agePct65up, and householdsize.



c. *[4 points]* Plot the squared error on the training and test data versus λ



d. *[4 points]* Sometimes a larger value of λ performs nearly as well as a smaller value, but a larger value will select fewer variables and perhaps be more interpretable. Inspect the weights (on features) for $\lambda = 30$. Which feature variable had the largest (most positive) Lasso coefficient? What about the most negative? Discuss briefly. A description of the variables in the data set can be found here: `http://archive.ics.uci.edu/ml/machine-learning-databases/communities/communities.names`.

Maximal value of w 0.0680 belongs to feature PctIlleg. PctIlleg is percentage of kids born to never married persons indicative of unstable home life and lack of support system. Following with the next highest weights we can see that percentage of persons living in dense housing (more than 1 person per room), number of vacant households and number of homeless people counted on the streets also highly correlate with total number of violent crimes. This indicates that violent crimes tend to be more common in poorer areas, with poor living conditions. This is telling us that is still possible to be unmarried with a kid, or a kid with unmarried parents, without committing a violent crime, as correlation is not necessarily causation.
Minimal value of w -0.0702 belongs to feature PctKids2Par. PctKids2Par is percentage of kids in family housing with two parents. The feature is indicative of stable home life, an existence of support system, both financial and emotional, for the child so it does not come as a surprise it anti-correlates with the total number of violent crimes per 100 000 population.

Feature Name	Magnitude
PctKids2Par	-0.07017508
PctHousOccup	-0.00724790
PctWorkMom	-0.00544586
agePct12t29	-0.00367240
LemasPctOfficDrugUn	0.00051549
PctVacantBoarded	0.00126885
pctUrban	0.01042646
MalePctDivorce	0.01102267
NumStreet	0.01575945
HousVacant	0.02049849
PctPersDenseHous	0.03062536
PctIlleg	0.06802481

e. *[4 points]* Suppose there was a large negative weight on agePct65up and upon seeing this result, a politician suggests policies that encourage people over the age of 65 to move to high crime areas in an effort to reduce crime. What is the (statistical) flaw in this line of reasoning? (Hint: fire trucks are often seen around burning buildings, do fire trucks cause fire?)
As mentioned above correlation is not necessarily causation. Moving elderly into violent crime areas would likely reduce some crime statistics that are not population-corrected but overall would not have a profound effect on crime, except perhaps providing criminals with easier-to-assault targets. That said, I have never seen a fire start but I have often seen fire trucks suspiciously close to one, so who knows.

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


def coordinate_descent(x, y, lambd, tolerance=0.001, initW=None, convergeFast=True):
    """Preforms coordinate coordinate descent Lasso algorithm on the given data.

    Parameters
    ----------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.
    lambd: `float`
        Regularization parameter.
    tolerance: `float`, optional
        Convergence is achieved when the convergence criterion is smaller than
        tolerance.
    convergeFast: `bool`, optional
        When True, the convergence is calculated as the difference between
        maximum of new weights and maximal value of old weights; implying that
        it's likelythe two actually point to the same feature. This is not
        optimal, bu is much faster than calculating absolute value of minimal
        difference of the old and the new weights, as it's possible to avoid
        storing the old weights. True by default.
    initW: `np.array`
        Initial weights, a vector of d feature weights.

    Returns
    -------
    w: `np.array`
        New feature weights estimates.
    """
    n, d = x.shape

    if initW is None:
        w = np.zeros(d)
    else:
        w = initW

    # precalculate values used in the loop in advance
    squaredX =  2.0 * x**2

    # ensure convergence is not met on first loop
    convergeCriterion = tolerance + 1
    while convergeCriterion > tolerance:
        if convergeFast:
            # not optimal test, but fast
            oldMax = w.max()
            deltas = []
        else:
            oldW = w.copy()

        # Algorithm 1 implementation
        b = np.mean(y - np.dot(x, w))
        for k in range(d):
            xk = x[:, k]
            ak = squaredX[:,k].sum()

            # ck sum must ignore k-th dimension so we set it to zero and use
            # dot product. This matches the definition of w too, so we can
            # leave it wth zero value unless smaller than -lambda or bigger
            # than lambda anyhow.
            w[k] = 0
            delta = 0
            ck = 2.0 * np.dot(xk, y - (b + np.dot(x, w)))

            if ck < -lambd:
                delta = (ck + lambd) / ak
                w[k] = delta
            elif ck > lambd:
                delta = (ck - lambd) / ak
                w[k] = delta

            if convergeFast:
                deltas.append(delta)
        if convergeFast:
            # Find maximum difference between iterations
            convergeCriterion = abs(oldMax-max(deltas))
        else:
            convergeCriterion = np.max(np.abs(oldW-w))

    return w


def generate_data(n, d, k, sigma):
    """Generates i.i.d. samples of the model:
        y_i = w^T x_i + eps
    where
        w_j = j/k if j in {1,...,k}
        w_j = 0 otherwise
    and epsilon is random Gussian noise with the given sigma and X are also
    drawn from a Normal distribution with sigma 1.

    Parameters
    ----------
    n : `int`
        Number of samples drawn at random from the model.
    d : `int`
        Dimensionality of the feature space.
    k : `int`
        Cutoff point after which elements of w are zero.

    Returns
    -------
    x : `np.array`
        n-by-d sized array of data.
    y : `np.array`
```

```
        Vector of n model values.
    """
    # gaussian noise and data
    eps = np.random.normal(0, sigma**2, size=n)
    x = np.random.normal(size=(n, d))

    # weights
    w = np.arange(1, d + 1) / k
    w[k:] = 0

    # labels
    y = np.dot(x, w) + eps
    return x, y


def plot(ax, x, y, label="", xlabel="", ylabel="", title="", xlog=True, lc='black', lw=2):
    """Plots a line on given axis.

    Parameters
    ----------
    ax: `matplotlib.pyplot.Axes`
        Axis to plot on.
    x: `np.array`
        X axis values
    y: `np.array`
        Y axis values
    label: `str`, optional
        Line label
    xlabel: `str`, optional
        X axis label
    ylabel: `str`, optional
        Y axis label
    title: `str`, optional
        Axis title
    xlog : `bool`, optional
        X axis scaling will be logarithmic
    lc : `str`, optional
        Line color
    lw: `int` or `float`, optional
        Line width

    Returns
    ------
    ax: `matplotlib.pyplot.Axes`
        Modified axis.
    """
    ax.set_title(title)
    ax.plot(x, y, label=label, color=lc, linewidth=lw)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    if xlog:
        ax.set_xscale('log')
    return ax


def lambda_max(x, y):
    """The smallest value of regularization parameter lambda for which the
    w is entirely zero.

    Parameters
    ----------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.

    Returns
    -------
    lambda: `float`
        Smallest lambda for which w is entirely zero.
    """
    return np.max(2 * np.abs(np.dot((y - np.mean(y)), x)))


def A4_setup(n=500, d=1000, k=100, sigma=1):
    """Creates data as instructed by A4 problem, calculates the smallest value
    of regularization parameter for which w is zero and returns data parameters,
    data and calculated lambda.

    Parameters
    ----------
    n : `int`, optional
        Number of samples drawn at random from the model. Default: 500
    d : `int`, optional
        Dimensionality of the feature space. Default: 1000
    k : `int`, optional
        Cutoff point after which elements of w are zero. Default: 100
    sigma: `float`, optional
        STD of the noise Gaussian distribution that is added to the data (see
        generate_data). Default: 1.

    Returns
    -------
    x : `np.array`
        The feature space. A matrix with n rows of feature vectors, each with d
        features.
    y : `np.array`
        A column vector of n model values.
    maxLambda: `float`
        Lambda for which w is zero everywhere.
    params: `dict`
        Dictionary of parameters used to create the data (n, d, k and sigma).
    """
    params = {'n': n, 'd': d, "k": k, "sigma": sigma}
    x, y = generate_data(n, d, k, sigma)
    maxLambda = lambda_max(x, y)
    return x, y, maxLambda, params
```

```
def A4(nIter=20, tolerance=0.001):
    """Sets the data up as instructed by problem A4 and runs coordinate
    descent Lasso algorithm nIter times, each time decreasing regularization
    parameter lambda by a factor of 1.5.
    Plots the number of non-zero-features against used lambda and false vs
    true discovery rates.

    Displays plots.

    Parameters
    ----------
    nIter: `int`, optional
        Number of different regularization parameter iterations to run. Default
        is 20.
    tolerance: `float`, optional
        Coordinate descent tolerance, sets convergence criteria (see coordinate_descent).
        Default: 0.001.
    """
    x, y, lambd, params = A4_setup()
    k = params['k']

    lambdas, numNonZeros, fdrs, tprs = [], [], [], []
    w = np.zeros(params['d'])
    for i in range(nIter):
        w = coordinate_descent(x, y, lambd, tolerance)

        nonZeros = np.count_nonzero(w)
        correctNonZeros = np.count_nonzero(w[:k])
        incorrectNonZeros = np.count_nonzero(w[k+1:])

        try:
            fdrs.append(incorrectNonZeros/nonZeros)
        except ZeroDivisionError:
            fdrs.append(0)
        tprs.append(correctNonZeros/k)

        lambdas.append(lambd)
        numNonZeros.append(nonZeros)

        lambd /= 1.5

    fig, axes = plt.subplots(1, 2, figsize=(10, 6))
    plot(axes[0], lambdas, numNonZeros, xlabel="Lambda",
         ylabel="Number of non zero features",
         title="N of non-zero features VS lambda.")
    plot(axes[1], fdrs, tprs, xlabel="False Discovery rate",
         ylabel="True Discovery rate", title="True VS False Discovery rates.",
         xlog=False)
    plt.show()


def A5_setup(trainFile="data/crime-train.txt", testFile="data/crime-test.txt"):
    """Reads the crime train and test data into a pandas dataframe and
    calculates maximum lambda value.

    Parameters
    ----------
    trainFile: `str`
        Path to file containing training data.
    testFile: `str`
        Path to file containing test data.

    Returns
    -------
    xTrain: `np.array`
        Training feature set.
    yTrain: `np.array`
        Label set, the response variable, the thing we model.
    lMaxTrain: `float`
        Maximal value of lambda for which all weights are zero.
    xTest: `np.array`
        Testing feature set.
    yTest: `np.array`
        Label set, the response variable, the thing we model.
    lMaxTest: `float`
        Maximal value of lambda for which all weights are zero.
    """
    train = pd.read_table(trainFile)
    test = pd.read_table(testFile)

    yTrain = train["ViolentCrimesPerPop"]
    xTrain = train.drop("ViolentCrimesPerPop", axis=1)
    lMaxTrain = lambda_max(xTrain, yTrain)

    yTest = test["ViolentCrimesPerPop"]
    xTest = test.drop("ViolentCrimesPerPop", axis=1)
    lMaxTest = lambda_max(xTest, yTest)

    return xTrain, yTrain, lMaxTrain, xTest, yTest, lMaxTest


def mean_square_error(x, y, w):
    """For given data x, response y and weights w calculates the mean square
    error of the model (the square of difference of predicted VS actual).

    Parameters
    ----------
    x: `np.array`
        Feature set.
    y: `np.array`
        Label set
    w: `np.array`
        Weights.

    Returns
    -------
    mse: `float`
        Mean square error.
    """
    a = y - np.dot(x, w)
```

```
    return (a.T @ a)/len(y)


def A5ab():
    """Sets the data up as instructed by problem A5 and runs coordinate
    descent Lasso algorithm untill the change in regularization parameter is
    smaller than 0.01. Each iteration decreases regularization parameter by a
    factor of 2.

    Plots the number of non-zero-features and mean square error against lambda.
    Displays plots.

    Parameters
    ----------
    nIter: `int`, optional
        Number of different regularization parameter iterations to run. Default
        is 20.
    tolerance: `float`, optional
        Coordinate descent tolerance, sets convergence criteria (see
        coordinate_descent). Default: 0.001.
    """
    xTrain, yTrain, lMaxTrain, xTest, yTest, lMaxTest = A5_setup()

    # initialize weights, rename lambda so we don't alter original
    n, d = xTrain.shape
    w = np.zeros(d)
    lambd = lMaxTrain

    colNames = xTrain.columns.tolist()
    idxAgePct12 = colNames.index('agePct12t29')
    idxPctSoc = colNames.index('pctWSocSec')
    idxPctUrban = colNames.index('pctUrban')
    idxAgePct65 = colNames.index('agePct65up')
    idxHouse = colNames.index('householdsize')

    wAgePct12, wPctSoc, wPctUrban, wAgePct65, wHouseholdsize = [] ,[], [], [], []
    lambdas, numNonZeros, sqrErrTrain, sqrErrTest = [], [], [], []

    # run the actual fit, note w overrides itself, do proper convergence
    # because this is much shorter loop than a.
    while lambd > 0.01:
        w = coordinate_descent(xTrain.values, yTrain.values, lambd, initW=w, convergeFast=False)

        numNonZeros.append(np.count_nonzero(w))
        lambdas.append(lambd)
        sqrErrTrain.append(mean_square_error(xTrain.values, yTrain.values, w))
        sqrErrTest.append(mean_square_error(xTest.values, yTest.values, w))
        wAgePct12.append(w[idxAgePct12])
        wPctSoc.append(w[idxPctSoc])
        wPctUrban.append(w[idxPctUrban])
        wAgePct65.append(w[idxAgePct65])
        wHouseholdsize.append(w[idxHouse])

        lambd /= 2.0

    fig1, ax1 = plt.subplots(figsize=(10, 6))
    plot(ax1, lambdas, numNonZeros, xlabel="Lambda",
         ylabel="Number of non zero features",
         title="N of non-zero features VS lambda.")

    c = plt.cm.viridis(np.linspace(0, 0.8, 5))
    fig2, ax2 = plt.subplots(figsize=(10, 6))
    plot(ax2, lambdas, wPctSoc, label="agePct12t29", lc=c[0])
    plot(ax2, lambdas, wPctSoc, label="pctWSocSec", lc=c[1])
    plot(ax2, lambdas, wPctUrban, label="pctUrban", lc=c[2])
    plot(ax2, lambdas, wAgePct65, label="agePct65up", lc=c[3])
    plot(ax2, lambdas, wHouseholdsize, label="Householdsize", xlabel="Lambda",
         ylabel="Weight value", title="Number of variables VS lambda.", lc=c[4])
    ax2.legend()

    fig3, ax3 = plt.subplots(figsize=(10, 6))
    plot(ax3, lambdas, sqrErrTrain, label="MSE Train", lc="gray")
    plot(ax3, lambdas, sqrErrTest, label="MSE Test", xlabel="Lambda",
         ylabel="Mean Square Error", title="Mean Square Error VS lambda.")
    ax3.legend()
    plt.show()


def A5cd():
    """Sets the data up as instructed by problem A5 and runs coordinate
    descent Lasso algorithm with regularization value of 30.

    Prints the weights and the names of the features with most positive, most
    negative value and a sorted table of features with non-zero values.

    Parameters
    ----------
    nIter: `int`, optional
        Number of different regularization parameter iterations to run. Default
        is 20.
    tolerance: `float`, optional
        Coordinate descent tolerance, sets convergence criteria (see
        coordinate_descent). Default: 0.001.
    """
    xTrain, yTrain, lMaxTrain, xTest, yTest, lMaxTest = A5_setup()

    # initialize weights, rename lambda so we don't alter original
    n, d = xTrain.shape
    w = np.zeros(d)
    lambd = 30

    lambdas, numNonZeros, sqrErr = [], [], []
    w = coordinate_descent(xTrain.values, yTrain.values, lambd, initW=w, convergeFast=False)
    numNonZeros.append(np.count_nonzero(w))
    lambdas.append(lambd)

    idxMaxW = w == max(w)
    idxMinW = w == min(w)
    print(f"Maximal value of w {w[idxMaxW][0]:.4f} belongs to feature {xTrain.columns[idxMaxW][0]}")
    print(f"Maximal value of w {w[idxMinW][0]:.4f} belongs to feature {xTrain.columns[idxMinW][0]}")
```

```
    print()
    print("Feature Name              |  Magnitude")
    print("-----------------------------------------")
    idxSorted = np.argsort(w)
    idxNonZeroSorted = np.nonzero(w[idxSorted])
    for feature, w_i in zip(xTrain.columns[idxSorted][idxNonZeroSorted], w[idxSorted][idxNonZeroSorted]):
        print(f"{feature:25} | {w_i:4.8f}")


def A5():
    """Runs A5ab and A5cd functions."""
    A5ab()
    A5cd()


if __name__ == "__main__":
    A4()
    A5()
```

Binary Logistic Regression *[30 points]*

A6. Let us again consider the MNIST dataset, but now just binary classification, specifically, recognizing if a digit is a 2 or 7. Here, let Y= 1 for all the 7's digits in the dataset, and use Y=-1 for 2. We will use regularized logistic regression. Given a binary classification dataset $(x_i, y_i)_{i=1}^n$ for $x_i \in \mathbb{R}$ dand $y_i \in -1, 1$ we showed in class that the regularized negative log likelihood objective function can be written a

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + e^{-y_i(b+x_i^T w)}\right) + \lambda||w||_2^2$$

Note that the offset term b is not regularized. For all experiments, use $\lambda = 10^{-1}$. Let $\mu_i(w,b) = \frac{1}{1+exp(-y_i(b+x_i^T w))}$

a. *[8 points]* Derive the gradients $\nabla_w J(w,b)$, $\nabla_b J(w,b)$ and give your answers in terms of $\mu_i(w,b)$ (your answers should not contain exponentials).
Write J over μ to get a more general expression:

$$\begin{aligned}
\nabla_w J(w,b) &= \nabla_w \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + e^{-y_i(b+x_i^T w)}\right) + \lambda||w||_2^2 \\
&= \frac{1}{n}\sum_{i=1}^{n} \nabla_w \left(\log \frac{1}{\mu_i(w,b)} + \lambda||w||_2^2\right) \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{-1}{\mu_i(w,b)\ln 10}\nabla_w \mu_i(w,b) + 2\lambda w
\end{aligned}$$

Solve $\nabla\mu_i$ for w and b via substitution and chain rule to get both required gradients:

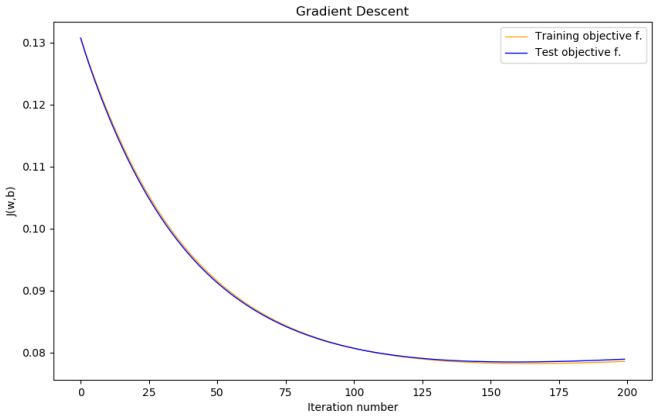
$$\begin{aligned}
\nabla_w \mu_i(w,b) &= \nabla_w \frac{1}{1+e^{-y_i b - y_i x_i^T w}} = \frac{-y_i x_i e^{-y_i b - y_i x_i^T w}}{-\left(1+e^{-y_i b - y_i x_i^T w}\right)^2} \\
&= \left(y_i x_i e^{-y_i b - y_i x_i^T w}\right)\mu_i^2 = y_i x_i \left(\frac{1}{\mu_i} - 1\right)\mu_i^2 \\
&= y_i x_i \mu_i (1-\mu_i) \\
\nabla_b \mu_i(w,b) &= \nabla_b \frac{1}{1+e^{-y_i b - y_i x_i^T w}} = \frac{-y_i e^{-y_i b - y_i x_i^T w}}{-\left(1+e^{-y_i b - y_i x_i^T w}\right)^2} \\
&= \left(y_i e^{-y_i b - y_i x_i^T w}\right)\mu_i^2 = y_i \left(\frac{1}{\mu_i} - 1\right)\mu_i^2 \\
&= y_i \mu_i (1-\mu_i)
\end{aligned}$$

Returning to J we have:

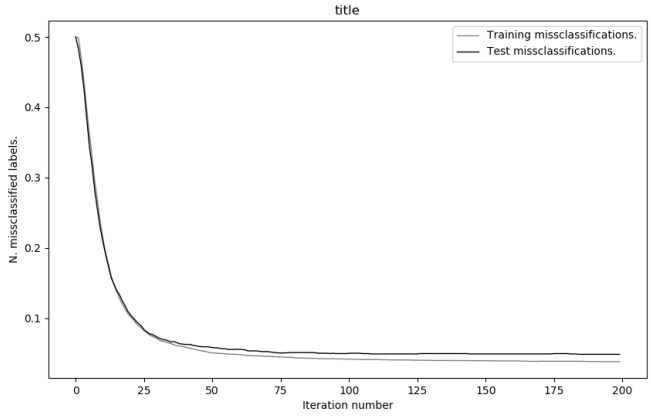
$$\begin{aligned}
\nabla_w J(w,b) &= \frac{-1}{n\ln 10}\sum_{i=1}^{n} \frac{1}{\mu_i(w,b)} y_i x_i \mu_i(1-\mu_i) + 2\lambda w \\
&= \frac{1}{n\ln 10}\sum_{i=1}^{n} y_i x_i(\mu_i - 1) + 2\lambda w \\
\nabla_b J(w,b) &= \frac{-1}{n\ln 10}\sum_{i=1}^{n} \frac{1}{\mu_i(w,b)} y_i \mu_i(1-\mu_i) \\
&= \frac{1}{n\ln 10}\sum_{i=1}^{n} y_i(\mu_i - 1)
\end{aligned}$$

b. *[8 points]* Implement gradient descent with an initial iterate of all zeros. Try several values of step sizes to find one that appears to make convergence on the training set as fast as possible. Run until you feel you are near to convergence.

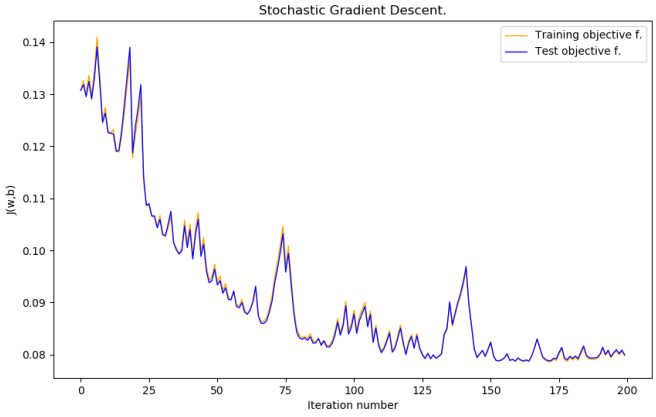
(a) For both the training set and the test, plot $J(w, b)$ as a function of the iteration number (and show both curves on the same plot)

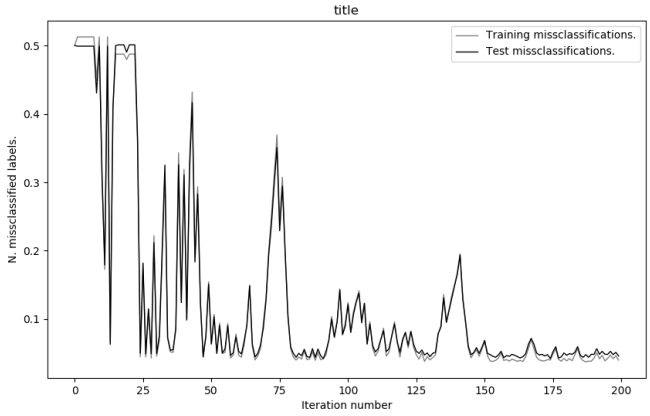


(b) For both the training set and the test, classify the points according to the rule $sign(b + x_i^T w)$ and plot the misclassification error as a function of the iteration number (and show both curves on the same plot). Note that you are only optimizing on the training set. The $J(w, b)$ and misclassification error plots should be on separate plots.

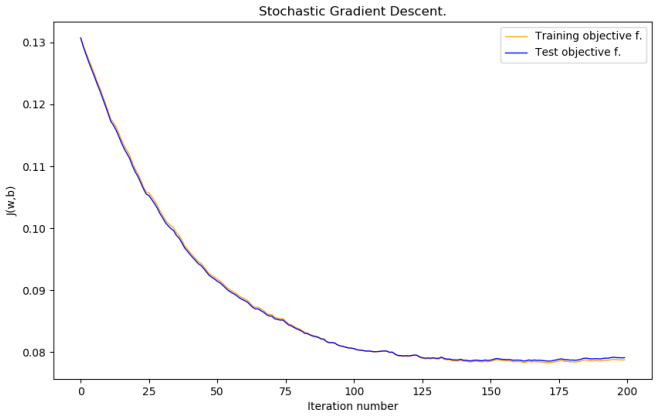


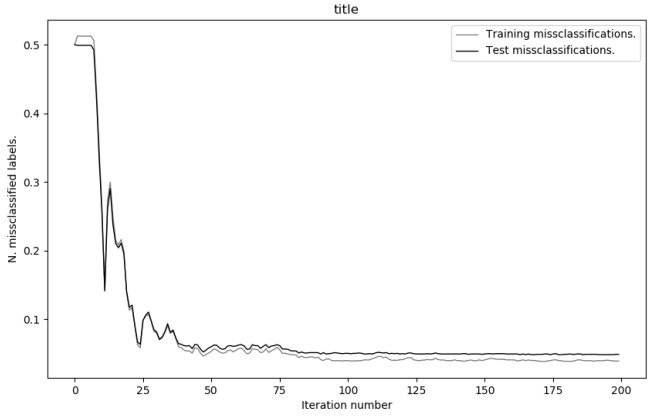
c. *[7 points]* Repeat (b) using stochastic gradient descent with a batch size of 1. Note, the expected gradient with respect to the random selection should be equal to the gradient found in part (a). Take careful note of how to scale the regularizer.





d. *[7 points]* Repeat (b) using stochastic gradient descent with batch size of 100. That is, instead of approximating the gradient with a single example, use 100. Note, the expected gradient with respect to the random selection should be equal to the gradient found in part (a).





```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import linalg

from mnist import MNIST


def separate(data, labels, *args):
    """Returns only data and labels that match the given values.

    Parameters
    ----------
    data: `np.array`
        Feature space data.
    labels: `np.array`
        Labels associated with the given features.
    args:
        Values of labels we want to separate data and labels on.

    Returns
    -------
    data: `np.array`
        The data for which labels match the given args.
    labels: `np.array`
        Labels that mathed the given args.
    """
    mask = [False]*len(labels)
    for arg in args:
        mask = np.logical_or(mask, labels == arg)
    return data[mask], labels[mask]


def encode_labels(labels, matchVals, encodeVals):
    """For given labels set values to elements of setVals where they match
    their respective matchVals.

    Parameters
    ----------
    matchVals: `tuple` or `list`
        Iterable of values labels must match in order to be set to their
        respective encoding values.
    encodeVals: `tuple` or `list`
        Encoding values assigned to matching values.
    """
    labels = labels.astype(int)
    for matchVal, encodeVal in zip(matchVals, encodeVals):
        labels[labels == matchVal] = encodeVal
    return labels


def mnist_numbers(path="data/mnist_data/", numbers=(2,7), encodeVals=(-1, 1)):
    """Loads MNIST data, located at path, normalizes, separates the desired
    numbers and encodes the matching data labels with given values.

    MNIST data are 28x28 pixel large images of letters from 0 to 9.

    Parameters
    ----------
    path: `str`
        path to the data directory
    numbers: `tuple` or `list`, optional
        Iterable of numbers that will be separated from the total dataset.
        Default: (2, 7)
    encodeVals: `tuple` or `list`, optional
        Encoding values assigned to each of the numbers separated.
        Default: (-1, 1)

    Returns
    -------
    train : `np.array`
        Train data normalized to 1
    trainlabels : `np.array`
        Encoded train data labels
    test : `np.array`
        Test data normalized to 1
    testLabels : `np.array`
        Encoded test data labels

    Notes
    -----
    The numbers and the encoding values must match in length and map according
    to their index. For example the default values encode number 2 labels as -1
    and number 7 labels as 1.
    Labels are encoded up to the shortest match to numbers, ignoring the rest.
    """
    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

    train, trainLabels = separate(train, trainLabels, *numbers)
    trainLabels = encode_labels(trainLabels, numbers, encodeVals)
    test, testLabels = separate(test, testLabels, *numbers)
    testLabels = encode_labels(testLabels, numbers, encodeVals)

    return train, trainLabels, test, testLabels


def J(x, y, w, b, lambd):
    """Calculates the regularized negative log likelihood function:

        J(w, b) = 1/n \sum log( 1/mu_i + lambda ||w||^2

    see `mu` for `mu_i(w, b)` for more.

    Parameters
```

```
    ----------
    x: `np.array`
        Feature space data.
    y: `np.array`
        Labels associated with the given features.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset
    lambd: `float`
        Regularization parameter.

    Returns
    -------
    J: `float`
        Objective, the regularized negative log likelihood.
    """
    n, d = x.shape
    exponential = np.exp(-(y*b + y*np.dot(x, w)))
    log = np.log10(1 + exponential) / (n*np.log(10))
    regularization = lambd * np.dot(w.T, w)
    return  np.sum(log) + regularization


def mu(x, y, w, b):
    """Calculates the value of the substitution expression:

        mu(w, b)  = 1 / ( 1+ exp(y(b + x^Tw)))

    that makes gradient calculations easier.

    Parameters
    ----------
    x: `np.array`
        Feature space data.
    y: `np.array`
        Labels associated with the given features.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset

    Returns
    -------
    mu : `float`
        Value of the substitution expression
    """
    exponential = np.exp(-y*b - y*np.dot(x, w))
    return 1 / (1+exponential)


def grad_w_J(x, y, w, b, lambd):
    """Calculates gradient of the regularized negative log likelihood function
    with respect to the weights:

        J(w, b) = 1/n \sum y_i x_i (1-mu_i) + 2 lambda w

    see `mu` for more on `mu_i`

    Parameters
    ----------
    x: `np.array`
        Feature space data.
    y: `np.array`
        Labels associated with the given features.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset
    lambd: `float`
        Regularization parameter.

    Returns
    -------
    grad_w_J: `float`
        Gradient of objective with respect to w
    """
    n, d = x.shape
    mus = mu(x, y, w, b) - 1
    # the trick to performing row-wise multiplication is to match the axis
    # sizes of the vectors and arrays by adding a dummy axis
    firstTerm = y[:, np.newaxis] * x * mus[:, np.newaxis]
    secondTerm = 2*lambd*w
    return np.sum(firstTerm, axis=0) / (n*np.log(10)) + secondTerm


def grad_b_J(x, y, w, b):
    """Calculates gradient of the regularized negative log likelihood function
    with respect to the offset:

        J(w, b) = 1/n \sum y_i (1-mu_i)

    see `mu` for more on `mu_i`

    Parameters
    ----------
    x: `np.array`
        Feature space data.
    y: `np.array`
        Labels associated with the given features.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset

    Returns
    -------
    grad_b_J: `float`
        Gradient of objective with respect to b
    """
```

```
    n, d = x.shape
    mus = mu(x, y, w, b) - 1
    # row-wise multiplication
    firstTerm = y[:, np.newaxis] * x * mus[:, np.newaxis]
    return np.sum(firstTerm) / (n*np.log(10))


def classify(x, w, b):
    """Returns binary classification of the data.

    Parameters
    ----------
    x: `np.array`
        Feature space we want classified.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset.

    Returns
    -------
    classes: `np.array`
        Array of binary positive or negative values (1 or -1) representing the
        classification of the model.
    """
    return np.sign(b + np.dot(x, w))


def count_missclassified(data, w, b, trueLabels):
    """Given features data, weights offsets and true labels counts the number
    of points missclassified by the model.

    Parameters
    ----------
    data: `np.array`
        Feature space to classify.
    w: `np.array`
        Model weights.
    b: `float`
        Model offset.
    trueLabels: `np.array`
        True labels for the features.

    Returns
    -------
    count: `int`
        Number of missclassified points.
    """
    return  np.sum(np.abs( classify(data, w, b) - trueLabels )) / (2*len(trueLabels))


def gradient_descent(data, labels, lambd, step, nIter=10, stochastic=False,  batchSize=1):
    """Performs gradient, or stochastic gradient, descent on the given data.

    Parameters
    ----------
    data: `np.array`
        Label space to learn the weights on.
    labels: `np.array`
        Labels for the given data.
    lambd: `float`
        Regularization parameter.
    step: `float`
        Steps size to take in the direction of the gradient.
    nIter: `int`, optional
        Number of iterations to preform, note that the function does not test
        for convergence so ensure sufficient number of steps. Default: 10
    stochastic: `bool`, optional
        If True preforms stochastic gradient descent. Default: False
    batchSize: `int`, optional
        Size of the data point set on which gradient estimate is performed on.
        Only used if stochastic is True. Default: 1

    Returns
    -------
    w: `np.array`
        Learned weights after nIter iterations.
    b: `float`
        Learned offsets after nIter iterations.
    calcJ: `np.array`
        Value of ojective in each step.
    missclassified: `np.array`
        Number of missclassified features in each step.
    wSteps: `np.array`
        Leaned weights in each step, used to estimate J and missclass. on test.
    bSteps: `np.array`
        Learned offsets in each step, used to estimate J and missclass. on test.
    """
    n, d = data.shape
    w = np.zeros(d)
    b = 0

    calcJ, missclassified, wSteps, bSteps = [], [], [], []
    for i in range(nIter):
        # append results befoe we alter values to capture zeroth element correctly
        wSteps.append(w)
        bSteps.append(b)
        calcJ.append(J(data, labels, w, b, lambd))
        missclassified.append(count_missclassified(data, w, b, labels))

        # stochastic picks some n random elements for the gradient calculation.
        # otherwise perform regular gradient estimate over all points
        if stochastic:
            idxs = np.random.permutation(np.arange(n))[:batchSize]
            w = w - step * grad_w_J(data[idxs], labels[idxs], w, b, lambd)
            b = b - step * grad_b_J(data[idxs], labels[idxs], w, b)
        else:
            w = w - step * grad_w_J(data, labels, w, b, lambd)
            b = b - step * grad_b_J(data, labels, w, b)
```

```
    return w, b, calcJ, missclassified, wSteps, bSteps


def plot(ax, x, y, label="", xlabel="", ylabel="", title="", xlog=False, lc='black', lw=1):
    """Plots a line on given axis.

    Parameters
    ----------
    ax: `matplotlib.pyplot.Axes`
       Axis to plot on.
    x: `np.array`
       X axis values
    y: `np.array`
       Y axis values
    label: `str`, optional
       Line label
    xlabel: `str`, optional
       X axis label
    ylabel: `str`, optional
       Y axis label
    title: `str`, optional
       Axis title
    xlog : `bool`, optional
       X axis scaling will be logarithmic
    lc : `str`, optional
       Line color
    lw: `int` or `float`, optional
       Line width

    Returns
    ------
    ax: `matplotlib.pyplot.Axes`
        Modified axis.
    """
    ax.set_title(title)
    ax.plot(x, y, label=label, color=lc, linewidth=lw)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    if xlog:
        ax.set_xscale('log')
    return ax


def A6abc(nIter, lambd, step, stochastic=False, batchSize=1, title="Gradient Descent"):
    """Reads MNIST dataset separates numbers 2 and 7, encodes -1 and 1 labels
    for the numbers respectively, performs gradient descent with the given
    parameters, estimates objective and missclassified labels in each step for
    both train and test datasets.

    Plots the values of objective and missclassifications for both train and
    test datasets.

    Parameters
    ----------
    nIter: `int`, optional
        Number of iterations to preform, note that the function does not test
        for convergence so ensure sufficient number of steps. Default: 10
    lambd: `float`
        Regularization parameter.
    step: `float`
        Steps size to take in the direction of the gradient.
    stochastic: `bool`, optional
        If True preforms stochastic gradient descent. Default: False
    batchSize: `int`, optional
        Size of the data point set on which gradient estimate is performed on.
        Only used if stochastic is True. Default: 1
    title: `str`
        Title of the produced plots.
    """
    trainData, trainLabels, testData, testLabels = mnist_numbers()

    # annoyingly we have to re-iterate or live with ugly grad_desc func.
    w, b, trainJ, testMisslbls, wSteps, bSteps =  gradient_descent(trainData, trainLabels, lambd,
                                                                   step, nIter=nIter, stochastic=stochastic,
                                                                   batchSize=batchSize)

    testJ, trainMisslbls = [], []
    for wi, bi in zip(wSteps, bSteps):
        testJ.append(J(testData, testLabels, wi, bi, lambd))
        trainMisslbls.append(count_missclassified(testData, wi, bi, testLabels))

    print(f"{title}")
    print(f"    Converged to offset b={b:.4f}")
    print(f"    Objective converged for train to J={trainJ[-1]:.4f} and test J={testJ[-1]:.4f}")

    iters = np.arange(nIter)

    fig1, ax1 = plt.subplots(figsize=(10, 6))
    ax1 = plot(ax1, iters, trainJ, label="Training objective f.", lc="orange")
    ax1 = plot(ax1, iters, testJ, label="Test objective f.", title=title,
               xlabel="Iteration number", ylabel="J(w,b)", lc="blue")
    ax1.legend()

    fig2, ax2 = plt.subplots(figsize=(10, 6))
    ax2 = plot(ax2, iters, testMisslbls, label="Training missclassifications.",
               lc="gray")
    ax2 = plot(ax2, iters, trainMisslbls, label="Test missclassifications.",
               title="title", xlabel="Iteration number",
               ylabel="N. missclassified labels.")
    ax2.legend()
    plt.show()


def A6b(nIter=200, lambd=0.1, step=0.01):
    """Calls A6abc with parameters specified in problem A6 b"""
    A6abc(nIter, lambd, step)


def A6c(nIter=200, lambd=0.1, step=0.01, stochastic=True, batchSize=1):
    """Calls A6abc with parameters specified in problem A6 c"""
```

```
    A6abc(nIter, lambd, step, stochastic, batchSize, title="Stochastic Gradient Descent.")


def A6d(nIter=200, lambd=0.1, step=0.01, stochastic=True, batchSize=100):
    """Calls A6abc with parameters specified in problem A6 d"""
    A6abc(nIter, lambd, step, stochastic, batchSize, title="Stochastic Gradient Descent.")


def A6():
    A6b()
    A6c()
    A6d()
    pass


if __name__ == "__main__":
    A6()
```