For the code provided as solutions to certain problems, I used the `numpy` library for python to easily represent matrices and vectors and be able to perform vector/matrix operations easily and quickly, as well as compute the norm of vectors/matrices, and the inverse of matrices, however I do not use any other features of the `numpy` library. All programs can be found at:

`https://github.com/JCRaymond/math5165/tree/master/homeworks/HW03`

(You can click on the link above, and should be able to see each of the programs mentioned below.)

**Problem 1.1.** Let $d, N, K \in \mathbb{N}$, and let $\{x_1, x_2, \dots, x_N\} \in \mathbb{R}^d$. The goal of $K$-means clustering is to find a collection $\{\mu_1, \mu_2, \dots, \mu_K\} \in \mathbb{R}^d\}$, and a set of indicator variables $r_{ij} \in \{0, 1\}$ for $i = 1, \dots, N$ and $j = 1, \dots, K$ such that for all $i$, $\sum_{j=1}^{K} r_{ij} = 1$ and

$$J = \sum_{i=1}^{N} \sum_{j=1}^{K} r_{ij} \|x_i - \mu_j\|_2^2,$$

the *distortion* is minimized. Using an alternating method of first minimizing $J$ over the $r_{ij}$, then over the $\mu_j$ should tend to a local minimum if a well behaved local minimum exists (a local minimum with an open set surrounding it such that it is the only local minimum in that open set). The alternating update steps are

$$r_{ij} = \begin{cases} 1, & j = \arg\min_k \|x_i - \mu_k\|_2^2 \\ 0, & \text{otherwise} \end{cases},$$

and

$$\mu_j = \frac{\sum_i r_{ij} x_i}{\sum_i r_{ij}}.$$

*Proof.* First, fix each $\mu_j$. Then, let $r_{ij}$ be a valid configuration satisfying the constraints such that for some $i_0$ and $j_0$, $r_{i_0 j_0} = 1$, however

$$j_0 \neq \arg\min_k \|x_{i_0} - \mu_k\|_2^2$$

in the sense that with

$$j' = \arg\min_k \|x_{i_0} - \mu_k\|_2^2,$$

$j_0$ is such that

$$\|x_{i_0} - \mu_{j_0}\|_2^2 > \|x_{i_0} - \mu_{j'}\|_2^2.$$

Since $r_{i_0 j_0} = 1$, it follows that $r_{i_0 j'} = 0$, since $j_0 \neq j'$ and $\sum_{j=1}^{K} r_{i_0 j} = 1$. By setting $r_{i_0 j_0} = 0$ and $r_{i_0 j'} = 1$, it is clear that $J'$, the resulting value of $J$ as specified above, would be

$$J' = J - \|x_{i_0} - \mu_{j_0}\|_2^2 + \|x_{i_0} - \mu_{j'}\|_2^2$$

where $J$ is the distortion corresponding to the original choice of $r_{ij}$. As a result, it follows that $J' < J$, and so no configuration of $r_{ij}$ satisfying the initial choice minimizes $J$. Therefore, it must be that

$$r_{ij} = r_{ij} = \begin{cases} 1, & j = \arg\min_k \|x_i - \mu_k\|_2^2 \\ 0, & \text{otherwise} \end{cases}.$$

Now, fix each $r_{ij}$. Note that since both sums in the distortion are finite, they can be swapped to give

$$J = \sum_{j=1}^{K} \sum_{i=1}^{N} r_{ij} \|x_i - \mu_j\|_2^2.$$

Each term in this double sum is nonnegative, and therefore minimizing $J$ over the $\mu_j$ is equivalent to minimizing each

$$J_j = \sum_{i=1}^{N} r_{ij} \|x_i - \mu_j\|_2^2$$

for $j = 1, \ldots, K$. Let $\mu_{jk}$ denote the $k$th component of the vector $\mu_j$, and similarly for $x_{ik}$ Then expanding the definition of the 2-norm,

$$J_j = \sum_{i=1}^{N} r_{ij} \sum_{k=1}^{d} (x_{ik} - \mu_{jk})^2 = \sum_{i=1}^{N} \sum_{k=1}^{d} r_{ij} x_{ik}^2 - 2 r_{ij} x_{ik} \mu_{jk} + r_{ij} \mu_{jk}^2.$$

Taking the derivative of this expression with respect to each $\mu_{jk}$ gives

$$\frac{\partial J_j}{\partial \mu_{jk}} = \sum_{i=1}^{N} -2 r_{ij} x_{ik} + 2 r_{ij} \mu_{jk} = 2 \left( \sum_{i=1}^{N} r_{ij} \right) \mu_{jk} - 2 \sum_{i=1}^{N} r_{ij} x_{ik}.$$

Furthermore,

$$\frac{\partial^2 J_j}{\partial \mu_{jk}^2} = 2 \left( \sum_{i=1}^{N} r_{ij} \right) \geq 0,$$

and therefore $J_j$ is convex, which means any zeroes of $\frac{\partial J_j}{\partial \mu_{jk}}$ are minima. Setting it equal to zero and manipulating gives

$$\mu_{jk} = \frac{\sum_{i=1}^{N} r_{ij} x_{ik}}{\sum_{i=1}^{N} r_{ij}},$$

which can be written as

$$\mu_j = \frac{\sum_{i=1}^{N} r_{ij} x_i}{\sum_{i=1}^{N} r_{ij}}$$

in general. As such, this rule for the update step will minimize $J$ over the set of $\mu_j$. □

**Problem 1.2.** The $K$-means clustering algorithm above can be applied to images by clustering colors together, and choosing the centroid of the corresponding pixel for the color of a new image. The result is an image with far fewer colors used.

For this, I decided to use the following image due to its colorful nature.


Figure 1: Original

In `p01-2.py`, I implemented the $K$-means algorithm specified in the previous part, and ran it on this image with $K \in \{2, 4, 8, 16, 32\}$. The results are as follows. The $K$-means algorithm ran to termination, meaning the colors chosen were in a local minimum of sort, and the clusters were stable.


Figure 2: 2 colors

As you can see, although pretty much all of the color is gone, the general image is still recognizable. Nevertheless, it seems too compressed.


Figure 3: 4 colors

With 4 colors however, the image becomes far more identifiable. The 4 colors allow for a lot of the major detail to come through, and for shading of objects to be possible, however there is not much minute details, say in the lamps. Rather than having 2 shades of brown, now there is a tanish-color, a brighter orange, then a brown and dark brown.


Figure 4: 8 colors

With 8 colors, more minutes details start coming through. Additionally, 2 shades of bluish-green are added, as well as additional shades of brown/orange. Still, in comparison with the original image, the shades of lighter green and purple are entirely missing.


Figure 5: 16 colors

The differences between 16 colors and 8 colors is rather subtle; one of the major differences is that there are more shades of each of the existing colors. The bluish-green shades have now become more blue, likely due to the addition of more true greys and blacks for aspects of the painting which are darker. The lighter green and purples in the original image are still not appearing, getting clustered with the blue colors.


Figure 6: 32 colors

The differences now are even more subtle. Largely, more shades of each of the existing colors have been added. A particular feature worth comparing between the images is the shading on the back of the bench. It's fairly clear that it gets more and more details as more clusters are used. In addition, some lighter shades of green start appearing in this final image, and it may be that if 64 colors were used, that we'd start to see more of the light greens and purples that appear in the leaves.

Overall, it's clear that the program worked, as it is easy to point out the specific colors used in the 2 and 4 cluster versions, and it is easy to see that the color-reduced images clearly resemble the original.

**Problem 2.1.** Let $f : \mathbb{R}^d \to \mathbb{R}$ be strictly convex and continuously differentiable with a global minimum. Then for any point $x \in \mathbb{R}^d$, the optimal learning rate $\alpha^*$ for the gradient descent step

$$x' = x - \alpha \nabla f(x)$$

is given by

$$\alpha^* = \arg\min_{\alpha} f(x - \alpha \nabla f(x)).$$

*Proof.* Let $x^*$ be the global minimum of $f$. With $x \in \mathbb{R}^d$ fixed, $\nabla f(x) \in \mathbb{R}^d$ is also fixed. This means we can define a subset $A = \{x - \alpha \nabla f(x) : \alpha \in \mathbb{R}\} \subset \mathbb{R}^d$ of possible values for $x'$, depending on the choice of $\alpha$. In the case that $x^* \in A$, then the optimal choice $\alpha^*$ is clearly such that

$$x^* = x - \alpha^* \nabla f(x),$$

however $x^*$ is not known, so it is not possible to calculate $\alpha^*$ using this equation. Instead, since $x^*$ minimizes $f$ on $\mathbb{R}^d$, and $x^* \in A$, $x^*$ also minimizes $f$ on $A$. Therefore, taking

$$\alpha^* = \arg\min_{\alpha \in A} f(x - \alpha \nabla f(x))$$

will result in

$$x^* = x' = x - \alpha^* \nabla f(x).$$

Now, suppose that $x^* \notin A$, and let $g : \mathbb{R} \to \mathbb{R}$ be defined by

$$g(\alpha) = f(x - \alpha \nabla f(x)).$$

Since $f$ is strictly convex, so is $g$, and since $f$ has a single global minimum, so should $g$. Let $\beta$ be a basis for $\mathbb{R}^d$ with $\beta_1 = \nabla f(x)$. Then the coordinate vector for $\nabla f(x)$ with respect to this basis is $(1, 0, \ldots, 0)$. Now, define the mapping $h : \mathbb{R} \to \mathbb{R}^d$ by

$$h(\alpha) = \big[\nabla f(x - \alpha \nabla f(x))\big]_\beta,$$

so that $h(\alpha)$ is the coordinate gradient of $f$ at the point $x - \alpha\nabla f(x)$ with respect to the basis $\beta$. Now, considering the first coordinate of $h(\alpha)$ for every $\alpha$, it must be that the absolute value of this coordinate is greater than the gradient of $h(\arg\min_\alpha g(\alpha))$. In fact, the coordinate for this choice of $\alpha$ should be 0. The reason is that along $A$, the gradient must have some component in the $\alpha\nabla f(x)$ due to strict convexity, however at the minimum, the gradient no longer must point in the $\alpha\nabla f(x)$ direction. Furthermore, since $f$ is convex, moving towards the least known value of $f$, such as by minimizing $g$, also moves towards the global minimum of $f$. □

**Problem 2.2.** This part is in regards to the implementation of gradient descent with a fixed step size and optimal step size, and running these on the Rosenbrock function with a variety of parameters, as well as the Himmelblau function. Everything is implemented in `p02.py`. After implementing gradient descent, both with a static step size and optimal step size, here are the following results starting from $x_0 = (0, 0)$, until the magnitude of the gradient is less than $10^{-7}$. For the static step size, the parameter $\alpha = 0.001$ was chosen.

For the Rosenbrock function, with $a = 2$, and a variety of values of $b$, here is the runtime and number of iterations (number of iterations capped at 1 million).

| $b$ | Static Time | Static Iters | Optimal Time | Optimal Iters |
|---|---|---|---|---|
| 0.1 | 1.37 | 195586 | 0.076 | 338 |
| 1 | 0.93 | 132120 | 0.11 | 485 |
| 10 | 0.90 | 125494 | 1.1 | 4775 |
| 25 | 0.91 | 125042 | 3.1 | 13093 |
| 50 | 0.94 | 124890 | 9.7 | 39609 |
| 100 | 7* | 1000000* | 31 | 129774 |

Interestingly, for the static step size, the number of iterations decreases until $b = 100$, though not by much at around $b = 50$. For $b = 100$, the constant step size did not converge within 1 million iterations. On the other hand, the optimal step size implementation took progressively more and more iterations to converge, however it does at least converge in the case of $b = 100$, whereas static step sizes were not able to converge in time.

Another interesting aspect is the time parameter. Although the optimal step size implementation consistently used less iterations, for $b \geq 10$, the optimal step implementation was consistently slower, due to the larger amount of work needed for each iteration. This may break down for $b \geq 100$, however, as for the static step implementation, the value of $f$ at $x$ was around 1.1, quite far from 0, so it may need to run for much longer than the optimal step implementation in order to find the minimum in this case.

As for the Himmelblau function, here are the same quantities:

| Static Time | Static Iters | Optimal Time | Optimal Iters |
|---|---|---|---|
| 0.0056 | 761 | 0.0072 | 26 |

Here, again, although the optimal step size implementation needed far fewer iterations in order to converge, it still took longer than the standard gradient descent implementation.

Overall, it seems like standard gradient descent may be best to use if the function is well behaved enough. However, if it is more complex, then the optimal step size can help to navigate the complexity and still find a minimum in a reasonable amount of time.

**Problem.** 3.1 The log of conditional likelihood function for $x^{(i)} \in \mathbb{R}^d$ and $y \in \{0, 1\}$ can be written as follows

$$LCL(x, y) = -\sum_{i=1}^{N} y^{(i)} \ln\big(p(x^{(i)})\big) + (1 - y^{(i)}) \ln\big(1 - p(x^{(i)})\big),$$

where

$$p(x^{(i)}) = \frac{e^{\alpha + \beta^\intercal x^{(i)}}}{1 + e^{\alpha + \beta^\intercal x^{(i)}}},$$

with $\alpha \in \mathbb{R}$, and $\beta \in \mathbb{R}^d$. The negative in front of the summation allows for this function to be minimized in order to use gradient descent. Then

$$\frac{\partial LCL(x,y)}{\partial \alpha} = \sum_{i=1}^{N} y^{(i)} - p(x^{(i)}),$$

and

$$\frac{\partial LCL(x,y)}{\partial \beta_j} = \sum_{i=1}^{N} \left[ y^{(i)} - p(x^{(i)}) \right] x_j^{(i)}.$$

*Proof.* First, note that

$$p(x^{(i)}) = \frac{e^{\alpha+\beta^\intercal x^{(i)}}}{1+e^{\alpha+\beta^\intercal x^{(i)}}} = \frac{1}{1+e^{-\alpha-\beta^\intercal x^{(i)}}} = (1+e^{-\alpha-\beta^\intercal x^{(i)}})^{-1},$$

and

$$1 - p(x^{(i)}) = \frac{1+e^{\alpha+\beta^\intercal x^{(i)}}}{1+e^{\alpha+\beta^\intercal x^{(i)}}} - \frac{e^{\alpha+\beta^\intercal x^{(i)}}}{1+e^{\alpha+\beta^\intercal x^{(i)}}} = (1+e^{\alpha+\beta^\intercal x^{(i)}})^{-1}.$$

Clearly, for any variable $y$,

$$\frac{\partial}{\partial y} 1 - p(x^{(i)}) = -\frac{\partial}{\partial y} p(x^{(i)}),$$

and so it suffices to find the derivatives for $p(x^{(i)})$ alone. With respect to $\alpha$,

$$\begin{aligned}
\frac{\partial}{\partial \alpha} p(x^{(i)}) &= -(1+e^{-\alpha-\beta^\intercal x^{(i)}})^{-2} \frac{\partial}{\partial \alpha}(1+e^{-\alpha-\beta^\intercal x^{(i)}}) \\
&= -p(x^{(i)})^2 e^{-\alpha-\beta^\intercal x^{(i)}} \frac{\partial}{\partial \alpha}(-\alpha-\beta^\intercal x^{(i)}) \\
&= p(x^{(i)}) \frac{e^{-\alpha-\beta^\intercal x^{(i)}}}{1+e^{-\alpha-\beta^\intercal x^{(i)}}} \\
&= p(x^{(i)}) \frac{1}{1+e^{\alpha+\beta^\intercal x^{(i)}}} \\
&= p(x^{(i)}) \big(1 - p(x^{(i)})\big),
\end{aligned}$$

and with respect to $\beta_i$,

$$\begin{aligned}
\frac{\partial}{\partial \beta_j} p(x^{(i)}) &= -(1+e^{-\alpha-\beta^\intercal x^{(i)}})^{-2} \frac{\partial}{\partial \beta_j}(1+e^{-\alpha-\beta^\intercal x^{(i)}}) \\
&= -p(x^{(i)})^2 e^{-\alpha-\beta^\intercal x^{(i)}} \frac{\partial}{\partial \beta_j}(-\alpha-\beta^\intercal x^{(i)}) \\
&= p(x^{(i)}) \frac{e^{-\alpha-\beta^\intercal x^{(i)}}}{1+e^{-\alpha-\beta^\intercal x^{(i)}}} x_j^{(i)} \\
&= p(x^{(i)}) \frac{1}{1+e^{\alpha+\beta^\intercal x^{(i)}}} x_j^{(i)} \\
&= p(x^{(i)}) \big(1 - p(x^{(i)})\big) x_j^{(i)}.
\end{aligned}$$

Taking the derivative of the complete $LCL$ with respect to $\alpha$ gives

$$\begin{aligned}
\frac{\partial LCL(x,y)}{\partial \alpha} &= -\sum_{i=1}^{N} y^{(i)} \frac{\partial}{\partial \alpha} \ln\big(p(x^{(i)})\big) + (1-y^{(i)}) \frac{\partial}{\partial \alpha} \ln\big(1-p(x^{(i)})\big) \\
&= \sum_{i=1}^{N} y^{(i)} \frac{1}{p(x^{(i)})} \frac{\partial}{\partial \alpha} p(x^{(i)}) - (1-y^{(i)}) \frac{1}{1-p(x^{(i)})} \frac{\partial}{\partial \alpha} p(x^{(i)}) \\
&= \sum_{i=1}^{N} y^{(i)} \frac{1}{p(x^{(i)})} p(x^{(i)})\big(1-p(x^{(i)})\big) - (1-y^{(i)}) \frac{1}{1-p(x^{(i)})} p(x^{(i)})\big(1-p(x^{(i)})\big) \\
&= \sum_{i=1}^{N} y^{(i)} \big(1-p(x^{(i)})\big) - (1-y^{(i)}) p(x^{(i)}) \\
&= \sum_{i=1}^{N} y^{(i)} - y^{(i)} p(x^{(i)}) - p(x^{(i)}) + y^{(i)} p(x^{(i)}) \\
&= \sum_{i=1}^{N} y^{(i)} - p(x^{(i)}).
\end{aligned}$$

Similarly, the derivative with respect to $\beta_j$ is

$$\begin{aligned}
\frac{\partial LCL(x,y)}{\partial \beta_j} &= -\sum_{i=1}^{N} y^{(i)} \frac{\partial}{\partial \beta_j} \ln\big(p(x^{(i)})\big) + (1-y^{(i)}) \frac{\partial}{\partial \beta_j} \ln\big(1-p(x^{(i)})\big) \\
&= \sum_{i=1}^{N} y^{(i)} \frac{1}{p(x^{(i)})} \frac{\partial}{\partial \beta_j} p(x^{(i)}) - (1-y^{(i)}) \frac{1}{1-p(x^{(i)})} \frac{\partial}{\partial \beta_j} p(x^{(i)}) \\
&= \sum_{i=1}^{N} y^{(i)} \frac{1}{p(x^{(i)})} p(x^{(i)})\big(1-p(x^{(i)})\big) x_j^{(i)} - (1-y^{(i)}) \frac{1}{1-p(x^{(i)})} p(x^{(i)})\big(1-p(x^{(i)})\big) x_j^{(i)} \\
&= \sum_{i=1}^{N} \Big[ y^{(i)} \big(1-p(x^{(i)})\big) - (1-y^{(i)}) p(x^{(i)}) \Big] x_j^{(i)} \\
&= \sum_{i=1}^{N} \big[ y^{(i)} - y^{(i)} p(x^{(i)}) - p(x^{(i)}) + y^{(i)} p(x^{(i)}) \big] x_j^{(i)} \\
&= \sum_{i=1}^{N} \Big[ y^{(i)} - p(x^{(i)}) \Big] x_j^{(i)}.
\end{aligned}$$

□

**Problem 3.2.** This problem includes the final parts for question 3. The implementations are all in `p03.py`.

The dataset I used for this problem comes from

`https://archive.ics.uci.edu/ml/datasets/adult`

and includes data about people, and whether or not they make more than \$50k a year. Of the attributes in the dataset, I used 7 of them: age, fnlwgt (based on the demographics of the population the person came from), education_num (a number representing the level of education, higher is more education), sex (0 is Male, 1 is Female), capital_gain, capital_loss, and hours worked per week. I could not use the other 7 attributes, since they could not easily be linearly encoded. This dataset contains 48842 data points, 32561 of which were in the training set, and 16281 in the test set. This split was already done and provided from the link.

I first normalize all of the data so that the minimum value of any attribute is 0, and the maximum value of any attribute is 1. This ensures that scaling is not an issue, and allows for the algorithm to consider each feature equally.

After running 100000 iterations, with a stochastic training percentage of 2% (on each iteration, only 2% of the training data was used in computing the gradient), and a learning rate of 0.02, gradient descent was able to obtain an 81.30% training accuracy, and an 81.10% testing accuracy. With the additional ridge term with weight 0.1, and all other parameters identical, the training accuracy was 75.92% and the test accuracy was 76.38%. As with other instances of a ridge regularization term, the training accuracy is lower than the test accuracy unlike without the ridge term, where the test accuracy was lower than the training accuracy.

In my testing, I observed that without the ridge regularization term, the weights would continue growing, however with the regularization term, the weights stayed relatively small. This might cause issue with longer runs, so having some sort of regularization term for logistic regression may be necessary in order to avoid the weights blowing up.

**Problem 4.1.** Let $A$ be an $m \times n$ matrix, and let $B$ be an $n \times p$ matrix. Let $\{p_i\}_{i=1}^n$ be a probability distribution, and let $c \in \mathbb{N}$. Let $C$ be random matrix produced by performing randomized matrix multiplication of $A$ and $B$ using $c$ samples according to the distribution $\{p_i\}_{i=1}^n$. Then

$$\mathbb{E}[C_{ij}] = (AB)_{ij},$$

and

$$\mathrm{Var}[C_{ij}] = \frac{1}{c}\sum_{n_l=1}^{n} \frac{A_{in_l}^2 B_{n_l j}^2}{p_k} - \frac{1}{c}(AB)_{ij}^2.$$

*Proof.* For one iteration $l \in \{1, \ldots, c\}$ of the random matrix multiplication algorithm, let $n_l$ denote the index of the column in $A$ and row in $B$ (denoted $A^{(n_l)}$ and $B_{(n_l)}$) such that the matrix

$$C^l = \frac{A^{(n_l)} B_{(n_l)}}{c p_{n_l}}$$

is the matrix added on step $l$. Specifically, the $i,j$th entry of this matrix is

$$C_{ij}^l = \frac{A_{in_l} B_{n_l j}}{c p_{n_l}}.$$

Since $n_l$ is randomly chosen in $\{1, \ldots, n\}$ according to $\{p_i\}_{i=1}^n$, then the expectation of $C_{ij}^l$ is

$$\mathbb{E}[C_{ij}^l] = \sum_{n_l=1}^{n} p_{n_l} \frac{A_{in_l} B_{n_l j}}{c p_{n_l}} = \frac{1}{c}(AB)_{ij}.$$

Additionally, using the LOTUS,

$$\mathbb{E}[(C_{ij}^l)^2] = \sum_{n_l=1}^{n} p_{n_l} \left(\frac{A_{in_l} B_{n_l j}}{c p_{n_l}}\right)^2 = \sum_{n_l=1}^{n} \frac{A_{in_l}^2 B_{n_l j}^2}{c^2 p_{n_l}}.$$

Now, according to the algorithm, $C = \sum_{l=1}^{c} C^l$, and so

$$\mathbb{E}[C_{ij}] = \sum_{l=1}^{c} \mathbb{E}[C^l] = (AB)_{ij}.$$

Then, the variance of $C_{ij}^l$ can be calculated to be

$$\mathrm{Var}[C_{ij}^l] = \mathbb{E}[(C_{ij}^l)^2] - \mathbb{E}[C_{ij}^l]^2 = \frac{1}{c^2}\sum_{n_l=1}^{n} \frac{A_{in_l}^2 B_{n_l j}^2}{p_{n_l}} - \frac{1}{c^2}(AB)_{ij}^2.$$

since each of the $C^l$ are independent (due to the $n_l$ being selected i.i.d.), the variance of $C$ is just the sum of the variances of the $C^l$. This gives

$$\mathrm{Var}[C_{ij}] = \sum_{l=1}^{c} \mathrm{Var}[C_{ij}^l] = \frac{1}{c}\sum_{n_l=1}^{n} \frac{A_{in_l}^2 B_{n_l j}^2}{p_{n_l}} - \frac{1}{c}(AB)_{ij}^2$$

□

**Problem 4.2.** Consider the same circumstance as the last problem. Then the distribution $\{p_i\}_{i=1}^n$ which minimizes

$$\mathbb{E}[\|AB - C\|_F^2],$$

where $\|\cdot\|_F$ denotes the Frobenius norm, is

$$p_i = \frac{\|A^{(i)}\|_2 \|B_{(i)}\|_2}{\sum_{k=1}^n \|A^{(k)}\|_2 \|B_{(k)}\|_2}.$$

*Proof.* First, by linearity of expectation and the definition of the Frobenius norm,

$$\begin{aligned}
\mathbb{E}[\|AB - C\|_F^2] &= \sum_{i=1}^m \sum_{j=1}^p \mathbb{E}[(AB - C)_{ij}^2] \\
&= \sum_{i=1}^m \sum_{j=1}^p \mathbb{E}[(\mathbb{E}[C] - C)_{ij}^2] \\
&= \sum_{i=1}^m \sum_{j=1}^p \mathrm{Var}[C_{ij}] \\
&= \sum_{i=1}^m \sum_{j=1}^p \frac{1}{c} \sum_{n_l=1}^n \frac{A_{in_l}^2 B_{n_l j}^2}{p_{n_l}} - \frac{1}{c}(AB)_{ij}^2 \\
&= \frac{1}{c} \sum_{n_l=1}^n \frac{1}{p_{n_l}} \left( \sum_{i=1}^m A_{in_l}^2 \right) \left( \sum_{j=1}^p B_{n_l j}^2 \right) - \frac{1}{c} \sum_{i=1}^m \sum_{j=1}^p (AB)_{ij}^2 \\
&= \frac{1}{c} \sum_{n_l=1}^n \frac{1}{p_{n_l}} \|A^{(n_l)}\|_2^2 \|B_{(n_l)}\|_2^2 - \frac{1}{c} \|AB\|_F^2
\end{aligned}$$

Since the second term does not rely on $\{p_i\}$, minimizing this expectation with respect to $\{p_i\}$ only involves minimizing

$$f(\{p_i\}) = \sum_{n_l=1}^n \frac{1}{p_{n_l}} \|A^{(n_l)}\|_2^2 \|B_{(n_l)}\|_2^2$$

subject to

$$\sum_{i=1}^n p_i = 1.$$

Using a Lagrange multiplier,

$$\mathcal{L}(\{p_i\}, \lambda) = \sum_{n_l=1}^n \frac{1}{p_{n_l}} \|A^{(n_l)}\|_2^2 \|B_{(n_l)}\|_2^2 + \lambda \left( \sum_{i=1}^n p_i - 1 \right).$$

Taking the derivative with respect to each $p_i$ gives

$$\frac{\partial}{\partial p_i} \mathcal{L}(\{p_i\}, \lambda) = -\frac{1}{p_i^2} \|A^{(i)}\|_2^2 \|B_{(i)}\|_2^2 + \lambda.$$

Setting this equal to 0, and solving for $p_i$ gives

$$p_i = \frac{\|A^{(i)}\|_2 \|B_{(i)}\|_2}{\sqrt{\lambda}}.$$

Since it must be the case that

$$1 = \sum_{i=1}^n p_i = \sum_{i=1}^n \frac{\|A^{(n_l)}\|_2 \|B_{(n_l)}\|_2}{\sqrt{\lambda}},$$

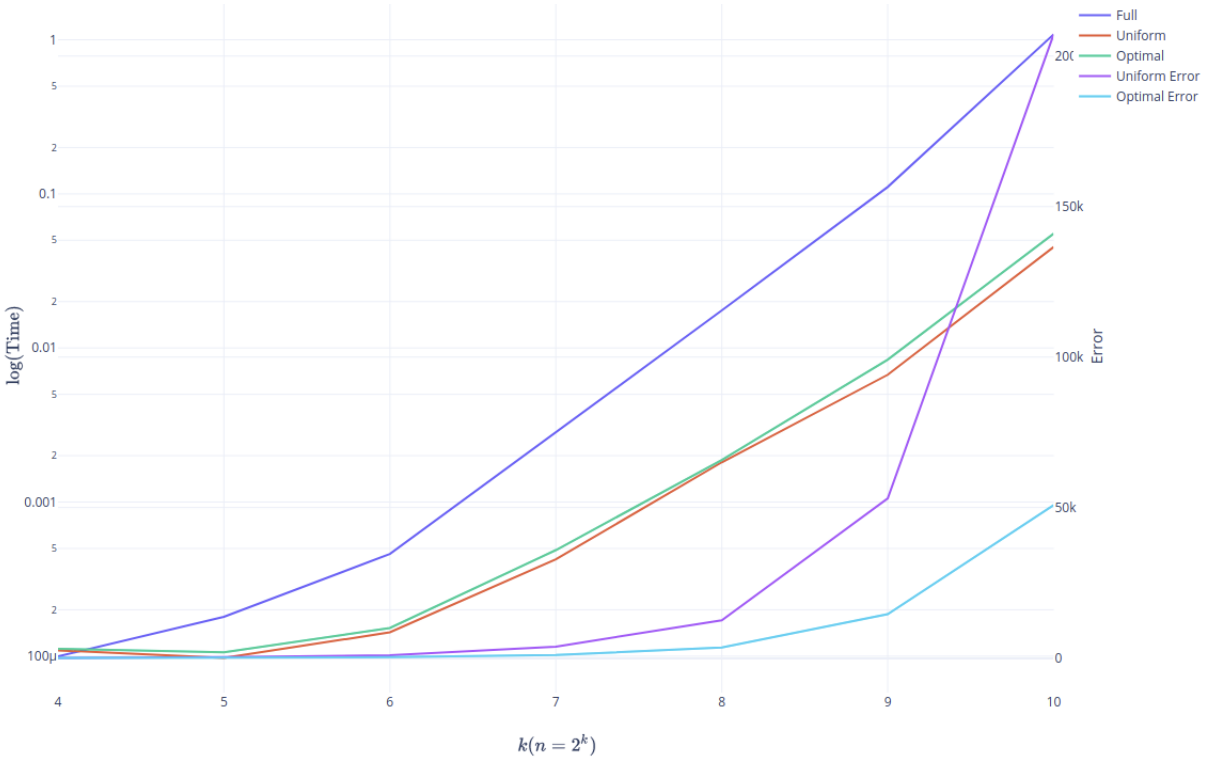
and so

$$\sqrt{\lambda} = \sum_{i=1}^{n} \|A^{(n_l)}\|_2 \|B_{(n_l)}\|_2.$$

Substituting this gives the optimal distribution for $\{p_i\}$. □

**Problem 4.3.** This last problem deals with the implementational details of random multiplication, which can be found in `p04.py`

After implementing randomized matrix multiplication as well as standard multiplication (so the comparison is apt, as `numpy's` implementation of multiplication is far faster than anything I can implement in Python), the results are summarized in this image:




First, consider the darker blue, orange, and light green lines (labeled Full, Uniform, and Optimal). These lines represent the average runtimes of performing 10 multiplications of random matrices, Full being the time it takes to perform a complete multiplication, Uniform being the time it takes to perform a randomized matrix multiplication with $c = \sqrt{n}$ samples and a uniform distribution, and Optimal being the time it takes to perform randomized matrix multiplication with $c = \sqrt{n}$ samples and the optimal distribution derived in the previous part. The $x$-axis is $k$, where each matrix being multiplied was $2^k \times 2^k$. On the left is the axis pertinent to these lines, a log(Time) scale. As you can see, performing the full matrix multiplication is consistently slower than the randomized matrix multiplication. In addition, it is clear that the rate that the time increases in $k$ is increasing, indicating that the randomized algorithm has a lower asymptotic time complexity. There is not much difference between the Uniform and Optimal runtimes, however the Optimal is consistently lower, likely due to the overhead required to compute the optimal distribution.

The last two lines, the purple and light blue ones (labeled Uniform Error and Optimal Error) indicates the average Frobenius norm of the difference between the result of random multiplication and full multiplication. The error quantity appears on the right hand $y$-axis labeled Error. Here, it is clear that the Optimal distribution produces a result far closer to the actual product of the matrices than the uniform distribution does.