CS 182: Introduction to Machine Learning, Fall 2022
Homework 3

(Due on Wednesday, Nov. 16 at 11:59pm (CST))

Notice:

- Please submit your assignments via Gradescope. The entry code is G2V63D.
- Please make sure you select your answer to the corresponding question when submitting your assignments.
- Each person has a total of five days to be late without penalty for all the assignments. Each late delivery less than one day will be counted as one day.

1. [20 points] [*SVM*]

   (a) In hard-margin SVM, the problem of maximizing margin can be converted into the following equivalent problem

$$\begin{aligned} \underset{\mathbf{w}, w_0}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) \geq 1, \ i = 1, \ldots, N. \end{aligned}$$

   (i) By introducing Lagrange multipliers $\{\alpha_i\}$, please give the Lagrangian function and the dual representation of the problem above. [6 points]

   (ii) Please show that the maximum of the margin $\gamma = \frac{1}{\|\mathbf{w}\|}$ is given by

$$\frac{1}{\gamma_{\max}^2} = \sum_{i=1}^{N} \alpha_i.$$

   (Hint: $\{\alpha_i\}$ can be obtained by solving the dual representation of the maximum margin problem.) [7 points]

   (b) The dual problem of soft-margin SVM is

$$\begin{aligned} \underset{\{\alpha_i\}}{\text{maximize}} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j r^{(i)}r^{(j)}\mathbf{x}^{(i)\mathsf{T}}\mathbf{x}^{(j)} \\ \text{subject to} \quad & \sum_{i=1}^{N} \alpha_i r^{(i)} = 0, \\ & 0 \leq \alpha_i \leq C, \ i = 1, \ldots, N, \end{aligned} \tag{1}$$

   where $\{\alpha_i\}$ are the dual variables and $(\mathbf{x}^{(i)}, r^{(i)}) \in \mathbb{R}^d \times \{-1, 1\}$ are feature-label pairs. Let $f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} + w_0$ be the prediction function, where $\mathbf{w} \in \mathbb{R}^d$, $b \in \mathbb{R}$ are primal variables. Argue from KKT conditions why the following hold:

$$\begin{aligned} \alpha_i = 0 &\Longrightarrow r^{(i)}f(\mathbf{x}^{(i)}) \geq 1, \\ 0 < \alpha_i < C &\Longrightarrow r^{(i)}f(\mathbf{x}^{(i)}) = 1, \\ \alpha_i = C &\Longrightarrow r^{(i)}f(\mathbf{x}^{(i)}) \leq 1. \end{aligned}$$

   [7 points]

Solution:

(a) (i) The primal Lagrangian is

$$\mathcal{L}(\mathbf{w}, w_0; \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N} \alpha_i \left(r^{(i)} \left(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0 \right) - 1 \right).$$

The dual representation of the maximum margin problem is

$$\begin{aligned} \underset{\boldsymbol{\alpha}}{\mathsf{maximize}} \quad & \widetilde{\mathcal{L}}(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j r^{(i)} r^{(j)} \mathbf{x}^{(i)\mathsf{T}}\mathbf{x}^{(j)} \\ \text{subject to} \quad & \alpha_i \geq 0, \ i = 1, \dots, N \\ & \sum_{i=1}^{N} \alpha_i r^{(i)} = 0. \end{aligned} \tag{2}$$

(ii) Start with the primal and write the KKT conditions: for $i = 1, \dots, N$,

$$\alpha_i \geqslant 0, \tag{3}$$

$$r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 \geqslant 0, \tag{4}$$

$$\alpha_i \left(r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 \right) = 0, \tag{5}$$

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i r^{(i)} \mathbf{x}^{(i)} \tag{6}$$

$$\sum_{i=1}^{N} \alpha_i r^{(i)} = 0. \tag{7}$$

Because $\gamma = \frac{1}{\|\mathbf{w}\|}$, maximizing γ is equal to minimizing $\|\mathbf{w}\|$. Meanwhile, the primal problem is a convex problem and Slater's condition is satisfied. Hence, the optimal solution must satisfy KKT conditon. According to (3), (4) and (5), when the primal problem reach its optimal value,

$$\mathcal{L}(\mathbf{w}, w_0; \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2. \tag{8}$$

Due to strong convexity, the optimal value of the original problem is equal to the optimal value of the dual problem. Therefore, combining the objective of (6), (6) and (8), we can get

$$\frac{1}{2}\|\mathbf{w}\|^2 = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j r^{(i)} r^{(j)} \mathbf{x}^{(i)\mathsf{T}}\mathbf{x}^{(j)} = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\|\mathbf{w}\|^2.$$

Therefore, we have

$$\frac{1}{\gamma_{\max}^2} = \|\mathbf{w}\|^2 = \sum_{i=1}^{N} \alpha_i.$$

(b) The primal Lagrangian is

$$\mathcal{L}(\mathbf{w}, w_0; \{\xi_i\}, \{\alpha_i\}, \{\mu_i\}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i \left(r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i \right) - \sum_{i=1}^{N} \mu_i \xi_i,$$

where $\mu_i \geq 0$ are the Lagrange multipliers introduced to enforce positivity of the ξ_i. The KKT conditions for the primal problem are therefore

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{N} \alpha_i r^{(i)} \mathbf{x}^{(i)} = 0 && (9)\\
\frac{\partial \mathcal{L}}{\partial w_0} &= -\sum_{i=1}^{N} \alpha_i r^{(i)} = 0 && (10)\\
\frac{\partial \mathcal{L}}{\partial \xi_i} &= C - \alpha_i - \mu_i = 0 && (11)\\
r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i &\geq 0 && (12)\\
\xi_i &\geq 0 && (13)\\
\alpha_i &\geq 0 && (14)\\
\mu_i &\geq 0 && (15)\\
\alpha_i \left(r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i \right) &= 0 && (16)\\
\mu_i \xi_i &= 0 && (17)
\end{aligned}$$

So we have

- $\alpha_i = 0 \overset{(11)}{\Longrightarrow} \mu_i = C \overset{(17)}{\Longrightarrow} \xi_i = 0 \overset{(12)}{\Longrightarrow} y_i f(\mathbf{x}_i) \geq 1$,
- $0 < \alpha_i < C \overset{(11)(16)}{\Longrightarrow} \mu_i > 0, r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i = 0 \overset{(17)}{\Longrightarrow} \xi_i = 0, r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i = 0 \Longrightarrow y_i f(\mathbf{x}_i) = 1$,
- $\alpha_i = C \overset{(16)}{\Longrightarrow} r^{(i)}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} + w_0) - 1 + \xi_i = 0 \overset{(13)}{\Longrightarrow} y_i f(\mathbf{x}_i) \leq 1$.

2. [20 points] [*SVR, kernel*] Given a dataset $\left\{(\mathbf{x}^{(i)}, r^{(i)})\right\}_{i=1}^{N}$. We wish to find a linear function f such that it can predict an input approximately correct, i.e, $f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} \approx r$ (already aborbed the bias, and $\mathbf{x} \in \mathbb{R}^d$). As in slides, we use the ϵ-insentive loss function $p_\epsilon(u) = \max(0, |u| - \epsilon)$. Accordingly, the SVR cost function is

$$J(\mathbf{w}) = \frac{1}{2}\left\|\mathbf{w}\right\|^2 + C\sum_{i=1}^{N} p_\epsilon(r^{(i)} - \mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}).$$

Here the term $\frac{1}{2}\left\|\mathbf{w}\right\|^2$ acts as a regularizer.

(a) By following essentially the same procedure as for SVM, write down the dual problem as a quadratic programming (QP). [13 points]

(b) Develop a kernelized version of the Dual Problem. Also specify how you may obtain the prediction for a new point $\mathbf{x}^{(t)}$. [5 points]

(c) How do you define "Support Vectors" for this problem? [2 points]

Solution:

(a) By introducing slack variables $s_i = \begin{cases} |r^{(i)} - f(\mathbf{x}^{(i)})| - \epsilon, & if\ |r^{(i)} - f(\mathbf{x}^{(i)})| > \epsilon \\ 0, & else \end{cases}$, we have the following problem

$$\begin{aligned} \underset{\mathbf{w}\in\mathbb{R}^d, \mathbf{s}\in\mathbb{R}^N}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} s_i \\ \text{subject to} \quad & s_i \geq 0,\ i = 1, \cdots, N \\ & |r^{(i)} - \mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}| \leq s_i + \epsilon,\ i = 1, \cdots, N. \end{aligned}$$

Let $\preceq$ and $\succeq$ denote elementwise inequalities, we obtain the following quadratic program:

$$\begin{aligned} \underset{\mathbf{w}\in\mathbb{R}^d, \mathbf{s}\in\mathbb{R}^N}{\text{minimize}} \quad & \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w} + C\mathbf{1}^\mathsf{T}\mathbf{s} \\ \text{subject to} \quad & -\mathbf{s} \preceq \mathbf{0}, \\ & -\mathbf{s} - \epsilon\mathbf{1} \preceq \mathbf{r} - \mathbf{X}\mathbf{w} \preceq \mathbf{s} + \epsilon\mathbf{1}. \end{aligned}$$

So we can get the Lagrangian as

$$\begin{aligned} \mathcal{L}(\mathbf{w}, \mathbf{s}; \boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \boldsymbol{\lambda}_3) &= \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w} + C\mathbf{1}^\mathsf{T}\mathbf{s} - \boldsymbol{\lambda}_1^\mathsf{T}\mathbf{s} + \boldsymbol{\lambda}_2^\mathsf{T}(\mathbf{r} - \mathbf{X}\mathbf{w} - \mathbf{s} - \epsilon\mathbf{1}) + \boldsymbol{\lambda}_3^\mathsf{T}(-\mathbf{s} - \epsilon\mathbf{1} - \mathbf{r} + \mathbf{X}\mathbf{w}) \\ &= \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w} + \left[\mathbf{X}^\mathsf{T}(\boldsymbol{\lambda}_3 - \boldsymbol{\lambda}_2)\right]^\mathsf{T}\mathbf{w} + (C\mathbf{1} - \boldsymbol{\lambda}_1 - \boldsymbol{\lambda}_2 - \boldsymbol{\lambda}_3)^\mathsf{T}\mathbf{s} + (\boldsymbol{\lambda}_2 - \boldsymbol{\lambda}_3)^\mathsf{T}\mathbf{r} - \epsilon\,(\boldsymbol{\lambda}_2 + \boldsymbol{\lambda}_3)^\mathsf{T}\mathbf{1}. \end{aligned}$$

We may derive the dual via the KKT Conditions. First compute the derivatives w.r.t to the primal variables.

$$\begin{aligned} \frac{\partial\mathcal{L}}{\partial\mathbf{w}} &= \mathbf{w} + \mathbf{X}^\mathsf{T}(\boldsymbol{\lambda}_3 - \boldsymbol{\lambda}_2) = \mathbf{0} \\ \frac{\partial\mathcal{L}}{\partial\mathbf{s}} &= C\mathbf{1} - \boldsymbol{\lambda}_1 - \boldsymbol{\lambda}_2 - \boldsymbol{\lambda}_3 = \mathbf{0} \end{aligned}$$

Let $\boldsymbol{\mu} = \boldsymbol{\lambda}_2 - \boldsymbol{\lambda}_3$. By setting the above to $\mathbf{0}$ and observing dual feasibilty, we have

$$\begin{aligned} &\boldsymbol{\lambda}_1 \succeq \mathbf{0} \Longrightarrow \boldsymbol{\lambda}_2 + \boldsymbol{\lambda}_3 \preceq C\mathbf{1} \Longrightarrow 2\boldsymbol{\lambda}_2 - \boldsymbol{\mu} \preceq C\mathbf{1}, \\ &\boldsymbol{\lambda}_3 \succeq \mathbf{0} \Longrightarrow \boldsymbol{\lambda}_2 \succeq \boldsymbol{\mu}. \end{aligned}$$

Accordingly, we have the following dual QP

$$\begin{aligned} \underset{\boldsymbol{\lambda}_2, \boldsymbol{\mu}}{\text{maximize}} \quad & -\frac{1}{2}\boldsymbol{\mu}^\mathsf{T}\mathbf{X}\mathbf{X}^\mathsf{T}\boldsymbol{\mu} + (\mathbf{r} + \epsilon\mathbf{1})^\mathsf{T}\boldsymbol{\mu} - 2\epsilon\boldsymbol{\lambda}_2^\mathsf{T}\mathbf{1} \\ \text{subject to} \quad & 2\boldsymbol{\lambda}_2 \preceq \boldsymbol{\mu} + C\mathbf{1}, \\ & \boldsymbol{\lambda}_2 \succeq \boldsymbol{\mu} \\ & \boldsymbol{\lambda}_2 \succeq \mathbf{0}. \end{aligned}$$

(b) To Kernelize the problem, we replace $\mathbf{X}\mathbf{X}^\top$ via a kernel matrix $\mathbf{K} = (k(x_i, x_j))_{ij} \in \mathbb{R}^{N\times N}$ and get

$$\begin{aligned} \underset{\boldsymbol{\lambda}_2, \boldsymbol{\mu}}{\mathsf{maximize}} \quad & -\frac{1}{2}\boldsymbol{\mu}^\top \mathbf{K}\boldsymbol{\mu} + (\mathbf{r} + \epsilon\mathbf{1})^\top \boldsymbol{\mu} - 2\epsilon\boldsymbol{\lambda}_2^\top \mathbf{1} \\ \mathsf{subject\ to} \quad & 2\boldsymbol{\lambda}_2 \preceq \boldsymbol{\mu} + C\mathbf{1}, \\ & \boldsymbol{\lambda}_2 \succeq \boldsymbol{\mu} \\ & \boldsymbol{\lambda}_2 \succeq \mathbf{0}. \end{aligned}$$

The prediction at a new point $\mathbf{x}^{(t)}$ is $\hat{f}(\mathbf{x}^{(t)}) = \sum_{i=1}^N \mu_i k(x_i, x_j)$. To see this, the prediction from the primal solution $\mathbf{w}$ is, $\hat{f}(\mathbf{x}^{(t)}) = \mathbf{w}^\top \mathbf{x}^{(t)}$. If we solve the dual problem, then $\hat{f}(\mathbf{x}^{(t)}) = \boldsymbol{\mu}^\top \mathbf{X}\mathbf{x}^{(t)}$. In the kernelized version, denote the mapping of $\mathbf{x}^{(t)}$ by $\boldsymbol{\varphi}(\mathbf{x}^{(t)})$ and the mapping of the training data by $\boldsymbol{\Phi}$. Then $\hat{f}(\mathbf{x}^{(t)}) = \boldsymbol{\mu}^\top \boldsymbol{\Phi}\boldsymbol{\varphi}(\mathbf{x}^{(t)})$ which can be computed using just the inner prducts as $\hat{f}(\mathbf{x}^{(t)}) = \sum_{i=1}^N \mu_i k(x_i, x_j)$.

(c) In classification, the support vectors are those points whose inequality constraints are active and are used in computing the prediction. Here, similarly they are the points for which $s_i > 0$ in the primal problem and hence by complementary slackness $\lambda_{1i} = 0 \iff 2\lambda_{2i} = \mu_i + C$ in the dual problem. Geometrically, these are the points that lie outside an ϵ-tube of the estimated function.

3. [20 points] [*Dimension Reduction*]

(a) Principal components analysis (PCA) reduces the dimensionality of the data by finding projection direction(s) that minimizes the squared errors in reconstructing the original data or equivalently maximizes the variance of the projected data. On the other hand, Fisher's linear discriminant is a supervised dimension reduction method, which, given labels of the data, finds the projection direction that maximizes the between-class variance relative to the within-class variance of the projected data. In the following Figure 1, draw the first principal component direction in the left figure, and the first Fisher's linear discriminant direction in the right figure (Note: for PCA, ignore the fact that points are labeled (as round, diamond or square) since PCA does not use label information. For linear discriminant, consider round points as the positive class, and both diamond and square points as the negative class). [7 points]

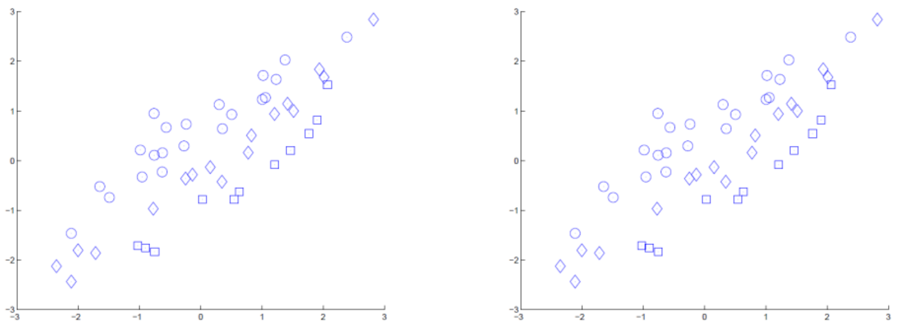


Figure 1: Draw the first principal component and linear discriminant component, respectively

(b) Canonical correlation analysis (CCA) handles the situation that each data point (i.e., each object) has two representations (i.e., two sets of features), e.g., a web page can be represented by the text on that page, and can also be represented by other pages linked to that page. Now suppose each data point has two representations $\mathbf{x}$ and $\mathbf{y}$, each of which is a 2-dimensional feature vector (i.e., $\mathbf{x} = [x_1, x_2]^T$ and $\mathbf{y} = [y_1, y_2]^T$). Given a set of data points, CCA finds a pair of projection directions $(\mathbf{u}, \mathbf{v})$ to maximize the sample correlation $\hat{\text{corr}}\left(\mathbf{u}^T\mathbf{x}\right)\left(\mathbf{v}^T\mathbf{y}\right)$ along the directions $\mathbf{u}$ and $\mathbf{v}$. In other words, after we project one representation of data points onto$\mathbf{u}$ and the other representation of data points onto $\mathbf{v}$, the two projected representations $\mathbf{u}^T\mathbf{x}$ and $\mathbf{v}^T\mathbf{y}$ should be maximally correlated (intuitively, data points with large values in one projected direction should also have large values in the other projected direction).
Now we can see data points shown in Figure 2, where each data point has two representations $\mathbf{x} = [x_1, x_2]^T$ and $\mathbf{y} = [y_1, y_2]^T$. Note that data are paired: each point in the left figure corresponds to a specific point in the right figure and vice versa, because these two points are two representations of the same object. Different objects are shown in different gray scales in the two figures (so you should be able to approximately figure out how points are paired). In the right figure we've given one CCA projection direction $\mathbf{v}$, draw the other CCA projection direction $\mathbf{u}$ in the left figure.[5 points]

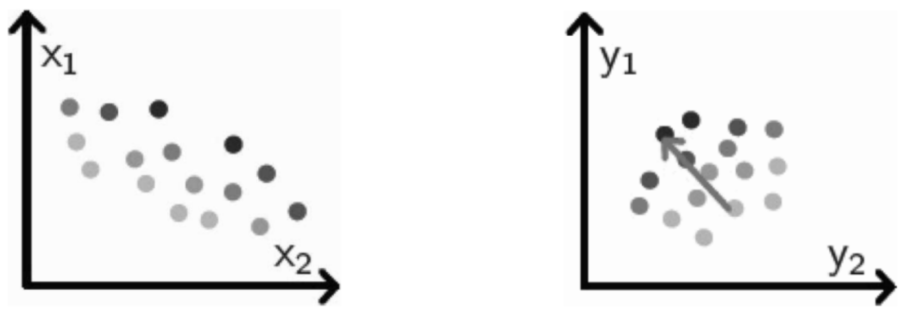


Figure 2: Draw the CCA projection direction in the left figure

(c) Consider 3 data points in the 2-d space:$(-1,-1), (0,0), (1,1)$. What is the first principal component (write down the actual vector)? Besides, If we project the original data points into the 1-d subspace by the principal component you choose, what are their coordinates in the 1-d subspace? And what is the variance of the projected data? [8 points]

Solution:

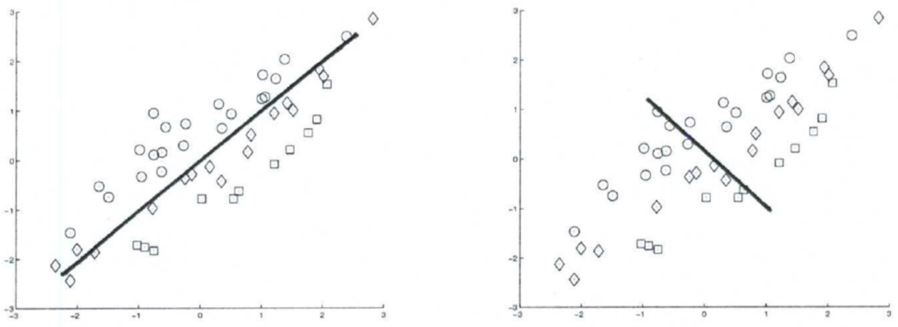


Figure 3: Solution of 3(a)

(a)

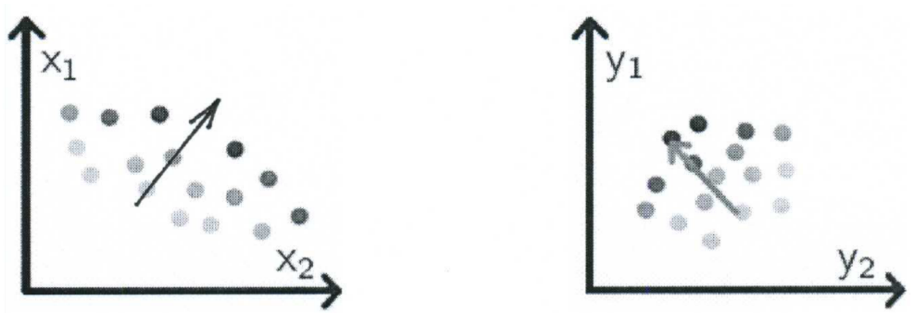


Figure 4: Solution of 3(b)

(b)

(c) The first principal component is $\mathbf{v} = \left[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]^T$. The coordinates of three points after projection should be $z_1 = \mathbf{x}_1^T\mathbf{v} = [-1,-1]\left[\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right]^T = -\sqrt{2}, z_2 = \mathbf{x}_2^T\mathbf{v} = 0, z_3 = \mathbf{x}_3^T\mathbf{v} = \sqrt{2}$. Note that the sample mean is 0, and thus the variance is $\frac{1}{3}\sum_{i=1}^{3}(z_i - 0)^2 = \frac{4}{3}$.

4. [20 points] [*Dimension Reduction*] Let $\mathbf{X}$ be a centered (i.e., zero-mean) sample matrix where each row is one observation and $\mathbf{\Sigma}$ be the sample covariance matrix. Assuming unit vector $\mathbf{w}_1$ is the eigenvector of $\mathbf{\Sigma}$ corresponding to the largest eigenvalue.

   (a) Prove that $\mathbf{w}_1$ is the principal component in principal component analysis (PCA) in that the projection onto direction $\mathbf{w}_1$ leads to the maximum variance for $\mathbf{X}$, i.e., $\mathbf{w}_1$ maximizes $\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}$ for any $\mathbf{w}$ satisfying $\|\mathbf{w}\|_2 = 1$. [10 points]

   (b) Suppose the sample $\mathbf{X}$ is labeled into two classes, briefly describe the idea of linear discriminant analysis (LDA) and discuss the similarities and differences between PCA and LDA. [5 points]

   (c) Show that $\mathbf{w}_1$ minimizes the reconstruction error $\left\|\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T\right\|_F^2$ for any $\mathbf{w}$ satisfying $\|\mathbf{w}\|_2 = 1$. [5 points]

Solution:

(a) The variance maximization problem is

$$\begin{array}{ll} \underset{\mathbf{w}}{\text{maximize}} & \mathbf{w}^T\mathbf{\Sigma}\mathbf{w} \\ \text{subject to} & \|\mathbf{w}\| = 1. \end{array}$$

By setting the Lagrangian's derivative to zero as follows:

$$\frac{\partial}{\partial \mathbf{w}} - \mathbf{w}^T\mathbf{\Sigma}\mathbf{w} + \lambda\left(\mathbf{w}^T\mathbf{w} - 1\right) = 0,$$

where λ is the Lagrange multiplier, leading to

$$\mathbf{\Sigma}\mathbf{w} = \lambda\mathbf{w}$$

Observe that

$$\mathbf{w}^T\mathbf{\Sigma}\mathbf{w} = \mathbf{w}^T\lambda\mathbf{w} = \lambda,$$

as $\mathbf{w}_1$ corresponds to the largest λ, we can conclude that $\mathbf{w}_1$ maximizes the variance $\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}$.

(b) LDA finds the vector $\mathbf{w}$ on which the data are projected such that the examples from the two classes are as well separated as possible.

Similarity of LDA and PCA: both are linear transformation techniques for dimension reduction.

Differences of LDA and PCA: LDA is a supervised technique that attempts to find a feature subspace that maximizes class separability, while PCA is unsupervised that focuses on capturing the direction of maximum variation in the data set.

(c) The reconstruction error minimization problem is

$$\begin{array}{ll} \underset{\mathbf{w}}{\text{maximize}} & f(\mathbf{w}) = \left\|\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T\right\|_F^2 \\ \text{subject to} & \|\mathbf{w}\| = 1. \end{array}$$

Observe that

$$\begin{aligned} f(\mathbf{w}) &= \left\|\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T\right\|_F^2 \\ &= \operatorname{tr}\left(\left(\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T\right)^T\left(\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T\right)\right) \\ &= \operatorname{tr}\left(\mathbf{X}^T\mathbf{X} - \mathbf{X}^T\mathbf{X}\mathbf{w}\mathbf{w}^T - \mathbf{w}_1\mathbf{w}^T\mathbf{X}^T\mathbf{X} + \mathbf{w}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\mathbf{w}^T\right) \\ &= \operatorname{tr}\left(\mathbf{X}^T\mathbf{X} + \mathbf{w}\mathbf{w}^T\mathbf{w}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\right) - 2\operatorname{tr}\left(\mathbf{w}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\right) \\ &= \operatorname{tr}\left(\mathbf{X}^T\mathbf{X}\right) - \operatorname{tr}\left(\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}\right). \end{aligned}$$

Therefore, it is obvious that the reconstruction error minimization problem is equivalent to the variance maximization problem, through which the proof is completed.

5. [20 points] [*Coding: MLP*] Complete "HW3-Coding.ipynb". After completion, you should convert your notebook to PDF, and concatenate the writing part and the coding part into one PDF which is the file to submit. Download the dataset from
`http://pan.shanghaitech.edu.cn/cloudservice/outerLink/decode?c3Vnb24xNjY3NzEzMzQ3ODg0c3Vnb24=`

Homework 3: Multi-Layer Perceptron (MLP)

In this exercise, you will implement a **two-layer neural network** to perform classification and test it on the CIFAR-10 dataset. [**20 points**]

1. Load the CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

We load the CIFAR-10 dataset from disk and perform preprocessing to prepare it for the two-layer neural network classifier.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from load_data import get_CIFAR10_data

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Training data shape:\t', X_train.shape)
print('Training labels shape:\t', y_train.shape)
print('Validation data shape:\t', X_val.shape)
print('Validation labels shape:', y_val.shape)
print('Test data shape:\t', X_test.shape)
print('Test labels shape:\t', y_test.shape)
```

```
Training data shape:     (49000, 3072)
Training labels shape:   (49000,)
Validation data shape:   (1000, 3072)
Validation labels shape: (1000,)
Test data shape:         (1000, 3072)
Test labels shape:       (1000,)
```

2. Implement a two-layer neural network

The two-layer neural network has the following architecture:

```
input - fully connected layer - ReLU - fully connected layer -
softmax
```

The outputs of the second fully-connected layer are the scores for each class.

We use the class `TwoLayerNet` in the file `two_layer_net.py` to represent instances of our network.

You need to fill the functions `loss`, `train`, and `predict` in the class `TwoLayerNet`. [**15 points**]

In [2]:
```python
import numpy as np


class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network. The net has an input dimension of
    N, a hidden layer dimension of H, and performs classification over C classes.
    We train the network with a softmax loss function and L2 regularization on the
    weight matrices. The network uses a ReLU nonlinearity after the first fully
    connected layer.

    In other words, the network has the following architecture:

    input - fully connected layer - ReLU - fully connected layer - softmax

    The outputs of the second fully-connected layer are the scores for each class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (D, H)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (H, C)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size:  The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        np.random.seed(128) # Do not change this
        self.params = {}
        self.params['W1'] = std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural
        network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] i
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
        the score for class c on input X[i].

        If y is not None, instead return a tuple of:
        - loss: Loss (data loss and regularization loss) for this batch of training
```

```
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameter
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        scores = None
        ############################################################################
        # TODO: Perform the forward pass, computing the class scores for the input.
        # Store the result in the scores variable, which should be an array of
        # shape (N, C).
        ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # define lamba function for relu
        relu = lambda x: np.maximum(0, x)

        # a1 = X x W1 = (N x D) x (D x H) = N x H
        a1 = relu(X.dot(W1) + b1)

        # scores = a1 x W2 = (N x H) x (H x C) = N x C
        scores = a1.dot(W2) + b2

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # If the targets are not given then jump out, we're done
        if y is None:
            return scores

        # Compute the loss
        loss = None
        ############################################################################
        # TODO: Finish the forward pass, and compute the loss. This should include
        # both the data loss and L2 regularization for W1 and W2. Store the result
        # in the variable loss, which should be a scalar. Use the Softmax
        # classifier loss.
        ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        scores -= scores.max(axis = 1, keepdims = True)

        # probs.shape is N x C
        probs = np.exp(scores)/np.sum(np.exp(scores), axis = 1, keepdims = True)

        # loss is a single number
        loss = np.sum(-np.log(probs[np.arange(N), y]))/N

        # Add regularization to the loss.
        loss += reg * (np.sum(W1 * W1) + np.sum(W2 * W2))

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # Backward pass: compute gradients
        grads = {}
        ############################################################################
        # TODO: Compute the backward pass, computing the derivatives of the weights
        # and biases. Store the results in the grads dictionary. For example,
        # grads['W1'] should store the gradient on W1, and be a matrix of same size
        ##########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # since dL(i)/df(k) = p(k) - 1 (if k = y[i]), where f is a vector of scores
        # i is the training sample and k is the class
        dscores = probs.reshape(N, -1) # dscores is (N x C)
        dscores[np.arange(N), y] -= 1

        # since scores = a1.dot(W2), we get dW2 by multiplying a1.T and dscores
        # W2 is H x C so dW2 should also match those dimensions
        # a1.T x dscores = (H x N) x (N x C) = H x C
        dW2 = np.dot(a1.T, dscores)/N

        # b2 gradient: sum dscores over all N and C
        db2 = dscores.sum(axis = 0)/N

        # since a1 = X.dot(W1), we get dW1 by multiplying X.T and da1
        # W1 is D x H so dW1 should also match those dimensions
        # X.T x da1 = (D x N) x (N x H) = D x H

        # first get da1 using scores = a1.dot(W2)
        # a1 is N x H so da1 should also match those dimensions
        # dscores x W2.T = (N x C) x (C x H) = N x H
        da1 = dscores.dot(W2.T)
        da1[a1 == 0] = 0 # set gradient of units that did not activate to 0
        dW1 = X.T.dot(da1)/N

        # b1 gradient: sum da1 over all N and H
        db1 = da1.sum(axis = 0)/N

        # Add regularization loss to the gradient
        dW1 += 2 * reg * W1
        dW2 += 2 * reg * W2

        grads = {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2}

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        return loss, grads

    def train(self, X, y, X_val, y_val,
              learning_rate=1e-3, learning_rate_decay=0.95,
              reg=5e-6, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning rate
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in range(num_iters):
            X_batch = None
            y_batch = None

            #########################################################################
            # TODO: Create a random minibatch of training data and labels, storing
            # them in X_batch and y_batch respectively.
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            indices = np.random.choice(num_train, batch_size)
            X_batch, y_batch = X[indices], y[indices]

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
            loss_history.append(loss)

            #########################################################################
            # TODO: Use the gradients in the grads dictionary to update the
            # parameters of the network (stored in the dictionary self.params)
            # using stochastic gradient descent. You'll need to use the gradients
            # stored in the grads dictionary defined above.
            #########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            self.params['W1'] -= learning_rate * grads['W1']
            self.params['W2'] -= learning_rate * grads['W2']
            self.params['b1'] -= learning_rate * grads['b1']
            self.params['b2'] -= learning_rate * grads['b2']

            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

            # Every epoch, check train and val accuracy and decay learning rate.
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

        return {
          'loss_history': loss_history,
          'train_acc_history': train_acc_history,
          'val_acc_history': val_acc_history,
        }

    def predict(self, X):
        """
        Use the trained weights of this two-layer network to predict labels for
        data points. For each data point we predict scores for each of the C
        classes, and assign each data point to the class with the highest score.

        Inputs:
        - X: A numpy array of shape (N, D) giving N D-dimensional data points to
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each of
          the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
          to have class c, where 0 <= c < C.
        """
        y_pred = None

        ###########################################################################
        # TODO: Implement this function; it should be VERY simple!                #
        ###########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        relu = lambda x: np.maximum(0, x)
        a1 = relu(X.dot(self.params['W1']) + self.params['b1'])
        scores = a1.dot(self.params['W2']) + self.params['b2']

        y_pred = np.argmax(scores, axis = 1)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        return y_pred


input_size = 3072
hidden_size = 80
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)
```

3. Train a network

We use SGD to train our network.

NOTE: Do not change preset parameters.

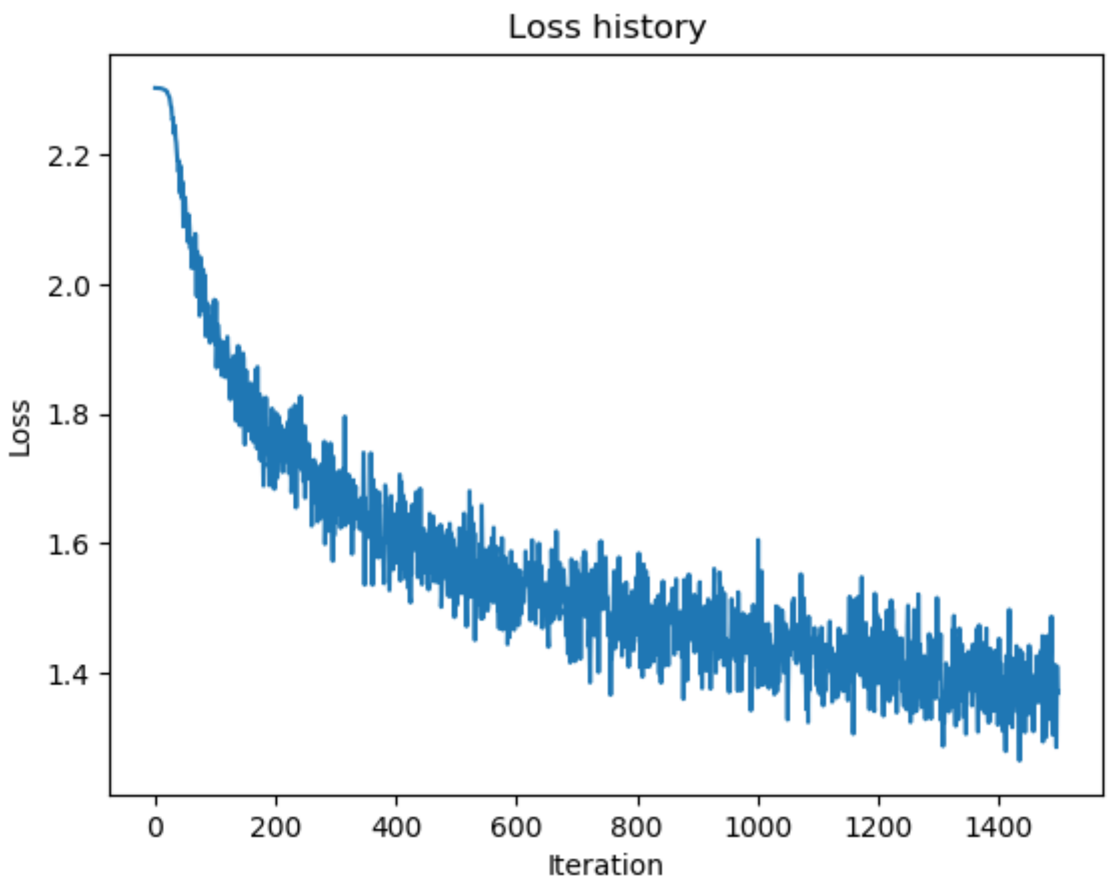
In [3]:
```
# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
                  learning_rate=1e-3, learning_rate_decay=0.95,
                  reg=1e-1, num_iters=1500, batch_size=490, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1500: loss 2.302817
iteration 100 / 1500: loss 1.908358
iteration 200 / 1500: loss 1.784551
iteration 300 / 1500: loss 1.707273
iteration 400 / 1500: loss 1.584537
iteration 500 / 1500: loss 1.549672
iteration 600 / 1500: loss 1.528404
iteration 700 / 1500: loss 1.527242
iteration 800 / 1500: loss 1.549146
iteration 900 / 1500: loss 1.469204
iteration 1000 / 1500: loss 1.378159
iteration 1100 / 1500: loss 1.453763
iteration 1200 / 1500: loss 1.383917
iteration 1300 / 1500: loss 1.422900
iteration 1400 / 1500: loss 1.355479
Validation accuracy:  0.505
```

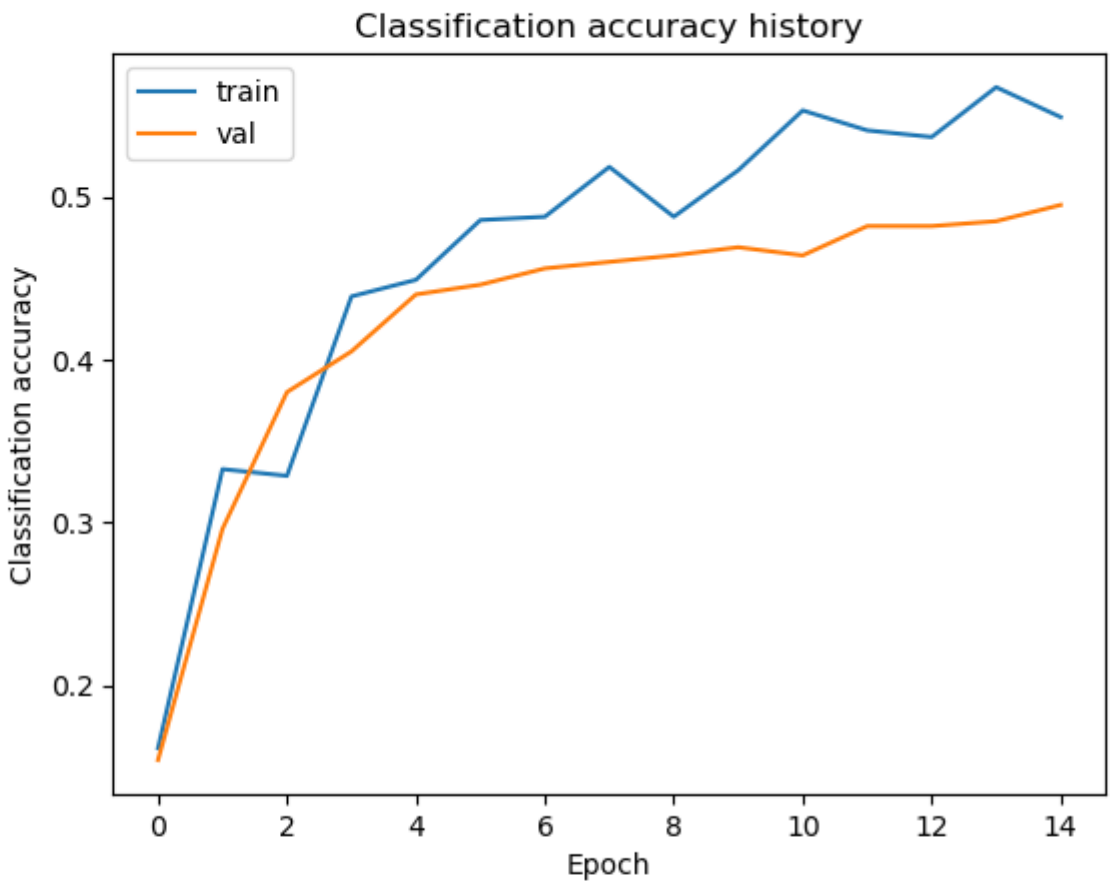
In [4]:

```python
# Plot the loss function
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```



In [5]:

```python
# Plot train / validation accuracies
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



Question:

What is the relationship between epoch, batch size, sample size and iteration? [**5 points**]

Answer:

[fill this]

Epoch = iteration * batch size / sample size

4. Run on the test set

In [6]:
```
test_acc = (net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
correct_test_acc = 0.502
print('difference: ', correct_test_acc-test_acc)
```

```
Test accuracy:  0.502
difference:  0.0
```

5. Export as PDF

Click `File` -> `Print`

NOTE: Please make sure that the submitted notebook has been run and the cell outputs are visible.