# CS 475 Machine Learning: Homework 5
# Dimensionality Reduction, Structured Prediction, and FATE

Due: Thursday December 5, 2019, 11:59pm

100 Points Total Version 1.0

**Make sure to read from start to finish before beginning the assignment.**

**Introduction** This assignment will consist of three parts:

- **(45 points)** - A coding portion, where you will implement two dimensionality reduction techniques: PCA and LLE. You will also implement a K-Nearest Neighbors classifier to operate over your dimensionality reduced data.
- **(15 points)** - A Jupyter notebook where you will explore bias in word embeddings, and implement a debiasing technique.
- **(40 points)** - Written questions about HMMs, CRFs, RNNs and FATE.

## 1 Programming (45 points)

### 1.1 K-Nearest Neighbor Classifier

You will implement a K-Nearest Neighbors Classifier (KNN). This is a very simple supervised model, which takes training data $x_i \in \mathcal{X}$, which is a finite set of $d$-dimensional vectors. Each $x_i$ is paired with a label $y_i$. "Training" a KNN is trivial: simply store the data. All of the "work" is done at prediction time. To predict a label for an example $x$

1. Find the $k$ nearest examples: $\mathcal{K}(x) = x_{(1)}, .., x_{(k)} \in \mathcal{X}$ where distance is measured by Euclidean distance to $x$.
2. Infer the label for $\widehat{y}$ as the most common label of $y_{(1)}, ..., y_{(k)}$, where $y_{(i)}$ is the label for $x_{(i)} \in \mathcal{K}(x)$. In case of a tie, select the label with the lowest integer index.

We have implemented the fit function for you (it just saves the data in the model object). Please implement the predict function for the `KNN` model in `models.py`.

You should be able to run this without specifying a `dr-algorithm` (dimensionality reduction algorithm), which will just run a KNN model over your raw data. For example, the command:

```
python3 main.py --knn-k 5 --test-split data/dev --predictions-file simple-knn-preds.txt
```

should output line-by-line predictions for the file `dev` into `simple-knn-preds.txt`, similar to past homeworks. You could then evaluate your predictions by running:

```
python3 accuracy.py --labeled data/labels-mnist-dev.npy --predicted simple-knn-preds.txt
```

Since KNN uses distance for prediction, larger dimensional data will be slower, and also suffer from the curse of dimensionality. We ask you to implement KNN as it will benefit from your dimensionality reduction.

For example, you could run your KNN model on the regular 748-dimension MNIST data. But this might take a while! We're going to attempt to fix that in the next two sections.

### 1.2 Principal Component Analysis

We discussed PCA as a dimensionality reduction technique in class. PCA is one of many different techniques, and the choice of method depends on the data and intended use of the reduced dimension dataset.

PCA relies on eigendecomposition, which factorizes a matrix into a canonical form consisting of two parts - eigenvalues and eigenvectors (see `https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix`). The eigenvectors determine the directions of the new feature space, and the eigenvalues determine their magnitude.

First, preprocess the data matrix: standardize each feature in the dataset to unit scale, defined as having a mean of zero and a variance of 1. Note that this is for each feature index, not the example or the entire dataset. You do this by subtracting the mean from all points and then dividing all points by the standard deviation. Since PCA yields a feature space that maximizes variance along the axes, it's important to ensure all features are measured in the same scale. (If the standard deviation is 0, which can happen in MNIST, you should not divide at all. This operation will leave you with every feature for this dimension being 0, which it probably was already.)

Second, calculate the covariance matrix $\Sigma$, which is a $D \times D$ matrix (where $D$ is the number of features). Each element of the matrix $\Sigma$ represents the covariance between two features. For a dataset of size $N$, each element in the matrix is defined as

$$\Sigma_{st} = \frac{1}{N-1} \sum_{i=1}^{N} (x_{is} - \bar{x}_s)(x_{it} - \bar{x}_t) \tag{1}$$

Third, we are now ready for PCA, which uses an eigensolver to perform eigendecomposition on the covariance matrix (see Section 1.2.1 for implementation details). For some square matrix $\mathbf{A}$, it can be factorized as

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-\mathbf{1}} \tag{2}$$

where $\mathbf{Q}$ are the eigenvectors and $\mathbf{\Lambda}$ are the eigenvalues.

Finally, select which eigenvectors to use in our lower-dimensional space by using the top $d$ corresponding eigenvalues (they will be sorted by absolute value). The eigenvalues indicate the amount of information the eigenvectors contain about the variance within the overall data. With these eigenvectors, we can build a weight vector $\mathbf{W}$, which is a $D \times d$ weight matrix. Finally, we can calculate our reduced feature space $\mathbf{Y}$,

$$\mathbf{Y} = \mathbf{X}\mathbf{W} \tag{3}$$

The result is a reduced dimension data matrix $\mathbf{Y}$ of size $N \times d$, where $N$ is the number of data points.

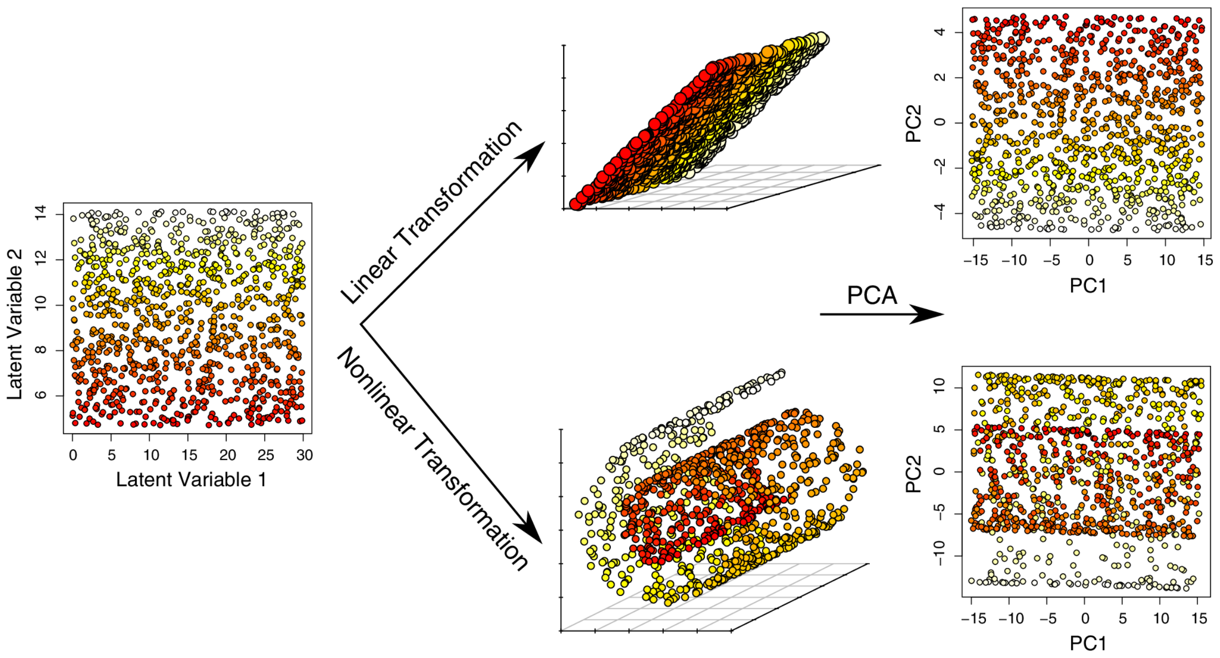

Figure 1: From https://link.springer.com/article/10.1007%2Fs11692-018-9464-9, an explanation of the linearity of PCA

#### 1.2.1 Implementation Details

Most of PCA can be implemented using straightforward numpy operations. The most challenging step is the eigendecomposition. We recommend using the scipy's eigensolver, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eig.html`. Additionally, you may use any function in either numpy or scipy.

You should implement PCA as a subclass of `Model`. It's `fit` function will take in both your training and test data and jointly reduce all of your data points together. Fit should then return the dimensionality reduced data *in the same order that it's passed in.* This is important because we will split the data back into training and test splits to perform KNN. You do not need to implement the `predict` function for this model.
Here is an example of how your model should run after you've implemented PCA:

```
python3 main.py --knn-k 5 --dr-algorithm pca --target-dim 300 --test-split dev \
    --predictions-file pca-knn-preds.txt
```

You should find this runs a lot faster than without dimensionality reduction!

### 1.3 Locally Linear Embedding (LLE)

While PCA strictly models linear variabilities in the data (see Figure 1), this strategy fails when the data lie on a low-dimensional "manifold" (`https://en.wikipedia.org/wiki/Manifold`), which can be a non-linear embedding of the data. Non-linear dimensionality reduction methods attempt to find this manifold and produce a low-dimensional space that encodes distances and relative position on the manifold.

Locally Linear Embedding (LLE) was introduced by Roweis and Saul in 2000. The original paper was published in Science, but you can find a more introductory tutorial here: `https://cs.nyu.edu/~roweis/lle/papers/lleintro.pdf`. The key insight to LLE is that if you have enough data points on a manifold, we can assume a point and its

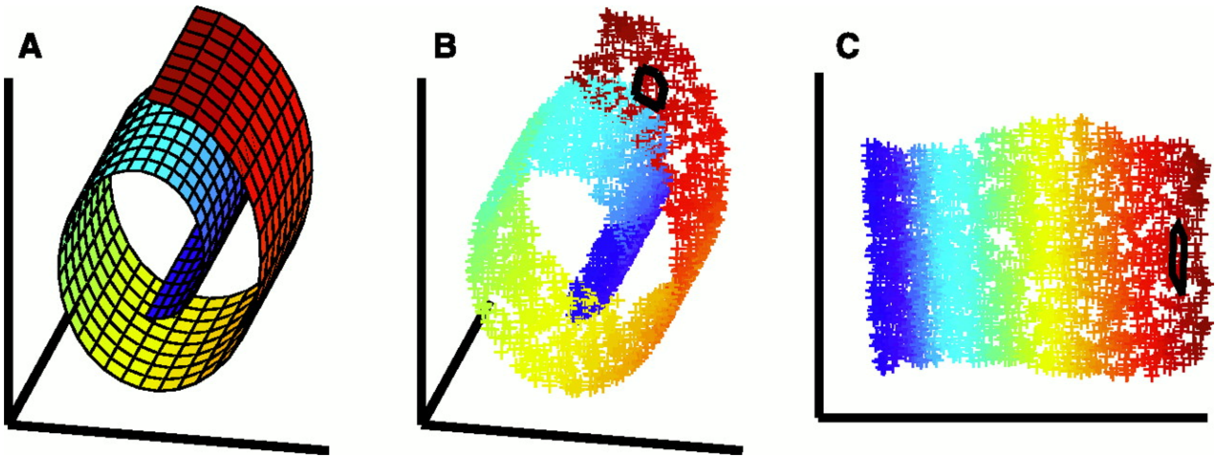

Figure 2: An illustration of LLE from `http://science.sciencemag.org/content/290/5500/2323`

nearest neighbors lie in a patch of the manifold that is approximately linear. Thus, LLE aims to reduce dimensionality by finding linear weights that best reconstruct each point from its neighbors, and then finding low-dimensional points for which these same weight matrices apply. LLE is illustrated in Figure 2.

You should begin by preprocessing your data to have features normalized with mean 0 and variance 1 as in PCA.

Then the LLE algorithm can be broken down into 3 steps:

1. For each data point $\mathbf{x}_i$, determine the $K$ nearest neighbors (where $K$ is a hyperparameter fixed ahead of time)

2. For each data point $\mathbf{x}_i$, compute the weights $w_{ij}$ that best reconstruct $\mathbf{x}_i$ from its neighbors (each neighbor is $\mathbf{x}_j$, and $\sum_j$ is a sum over neighbors of $\mathbf{x}_i$), using the reconstruction error function

$$\varepsilon(\mathbf{W}) = \sum_i |\mathbf{x}_i - \sum_j w_{ij}\mathbf{x}_j|^2 \tag{4}$$

3. Compute the lower-dimensional vectors $\mathbf{y}_i$ that are best reconstructed by the $w_{ij}$ computed in step 2, using an eigensolver to find the bottom nonzero eigenvectors to minimize the embedding cost function (same form as in step 2, but now $w_{ij}$ fixed instead of free)

$$\Phi(\mathbf{Y}) = \sum_i |\mathbf{y}_i - \sum_j w_{ij}\mathbf{y}_i|^2 \tag{5}$$

We use $\mathbf{X}$ to refer to the input data matrix, which consists of $N$ columns of data points of dimensionality $D$. We will be returning a low-dimensional embedding matrix $\mathbf{Y}$, which is $d$ x $N$ where $d \ll D$. $\mathbf{x}_i$ and $\mathbf{y}_i$ refer to the $i^{\text{th}}$ column of $\mathbf{X}$ and $\mathbf{Y}$ respectively, which corresponds to the $i^{\text{th}}$ data point. $K$ is the number of neighbors we will consider. $\mathbf{W}$ is the reconstruction weight matrix, and $w_{ij}$ is the entry corresponding to the $i^{\text{th}}$ row and $j^{\text{th}}$ column.

We now present the pseudocode, in the same three high-level steps outlined above.[1] We use for loops for simplicity of pseudocode, but we encourage you to consider numpy speed-ups.

1. Find nearest neighbors

```
for $i \in 1 \dots N$ do
    Keep track of the indices of the $K$ nearest neighors as an array $n_i^{(1)}, \dots n_i^{(k)}$
    for $j \in 1 \dots N$ do
        Calculate the Euclidean distance between $\mathbf{x}_i$ and $\mathbf{x}_j$
        if $\mathbf{x}_j$ is in $K$ nearest then
            Assign $j$ as a neighbor of $i$
        end if
    end for
end for
```

2. Solve for reconstruction weights

```
Initialize reconstruction weight matrix $\mathbf{W}$ as $N$ x $N$ matrix of zeros.
for $i \in 1 \dots N$ do
    Initialize $\mathbf{Z}$ as $D$ x $K$ matrix.
    for $k \in 1 \dots K$ do
        Set the $k^{\text{th}}$ column of $\mathbf{Z}$ to be $\mathbf{x}_{n_i^{(k)}} - \mathbf{x}_i$
    end for
    Compute local covariance $\mathbf{C} = \mathbf{Z}^T\mathbf{Z}$
    Regularization term $\epsilon = 1e^{-3} \times trace(\mathbf{C})$
    $\mathbf{C} = \mathbf{C} + \epsilon \times I$
    Solve the linear system $\mathbf{C}\widehat{\mathbf{w}} = \mathbf{1}$ for $\widehat{\mathbf{w}}$
    for $k \in 1 \dots K$ do
        $w_{in_i^{(k)}} = \widehat{w}_k / \text{sum}(\widehat{\mathbf{w}})$ where $\widehat{w}_k$ is the $k^{\text{th}}$ element of $\widehat{\mathbf{w}}$
    end for
end for
```

3. Compute embedding coordinates

```
Create sparse matrix $\mathbf{M} = (\mathbf{I} - \mathbf{W})^T(\mathbf{I} - \mathbf{W})$
Find bottom $d + 1$ eigenvectors of $M$ (corresponding to the smallest eigenvalues)
Remove the eigenvector with eigenvalue 0
Set the rows of $\mathbf{Y}$ to be the remaining $d$ eigenvectors
```

When we refer to "bottom eigenvectors," we mean the eigenvectors with the smallest eigenvalues. "Top eigenvectors" have the largest. We can get these easily from any standard eigensolver, which we'll detail more in the next section. Because $\mathbf{M}$ is symmetric, it will have only real eigenvalues and eigenvectors, so you don't have to worry about complex numbers here.

[1]Our pseudocode is adapted from the authors' web page, `https://cs.nyu.edu/~roweis/lle/algorithm.html`.

### 1.3.1 Implementation Details

i Just as a small tip, LLE can take a while to train on all 3,000 train examples + 1,000 test examples. It might be a good idea to reduce it down to 1,000 examples total while you're developing the algorithm to help you debug.

ii Because we're working with a symmetric matrix, you'll want to use the eigensolver provided with scipy here: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigsh.html#scipy.sparse.linalg.eigsh`. Because we're looking for the eigenvectors corresponding to the smallest eigenvalues, we'll want to use the flag `sigma=0.0` (this seems to work better than `which='sm'`).

iii Because the eigensolver is a numerical approximation, it is not guaranteed the recover exact zeros in the bottom eigenvalues. You may use `np.finfo(float).eps` as the lower bound.

iv Just as you implemented a subclass of `Model` for PCA, you'll do the same for LLE with the same behavior.

v It is not always trivial to decide $k$ (the number of nearest neighbors) in LLE. For this assignment, you can use $k = 10$. You should feel free to play with this, but this setting should work.

As before, here is an example of how we might run your code

```
python3 main.py --knn-k 5 --dr-algorithm lle --lle-k 10 --target-dim 300 --test-split dev \
    --predictions-file pca-knn-preds.txt
```

## 1.4 Programming Details

As before, your code will be evaluated on it's predictions on held-out test data. We have provided you with the test-data, so you can ensure that your code does not fail when running over it, but not the labels. You may compare the scores of your classifier using different dimensionality reduction techniques on development data with your peers on Piazza. To make sure that your predictions output is in the correct format, you should test that `accuracy.py` can run on your dev predictions, as described in the KNN instructions.

We will evaluate your code in 2 settings (1) classification accuracy on MNIST test data when using PCA dimensionality reduction, and (2) classification accuracy on MNIST test data when using LLE dimensionality reduction.

### 1.4.1 Data

You will be working with a subset of the popular MNIST [2] digit recognition dataset. This dataset is a multiclass classification dataset which is composed of 10 classes (0-9). Each example is a 28 by 28 greyscale image of a handwritten digit (0-9). Our version of the dataset comes in 3 splits, of which you have 5 files: `mnist-train.npy, mnist-dev.npy, mnist-test.npy, labels-mnist-train.npy, labels-mnist-dev.npy`

The training set consists of 3,000 labeled examples of hand written digits. The dev and test splits each contain 1,000 examples. While this is not a lot of training data (the actual MNIST dataset has 60,000 examples), we've cut things down considerably to help

[2] `https://en.wikipedia.org/wiki/MNIST_database`

you run your code faster. You might be surprised at the accuracy of a simple KNN model in this setting. Your implementations should not take longer than 7-8 minutes to run with `knn-k=5` and `lle-k=10` (LLE will be much longer runtime than PCA).

Additionally, your data should contain `glove.6B.100d.txt`, which you will use for the Notebook portion of this assignment.

### 1.4.2 Command Line Arguments

Your code should contain the following command line arguments, which you should not change:

- `--data`: Points to the directory containing the MNIST data splits.
- `--predictions-file`: The file that your predictions will be stored in.
- `--test-split`: The split (test or dev) that your KNN model will predict over.
- `--knn-k`: A hyperparameter for your KNN model. How many nearest neighbors to consider during prediction.
- `--dr-algorithm`: The type of dimensionality reduction algorithm to use (pca or lle).
- `--target-dim`: The number of dimensions to keep with your dr technique.
- `--lle-k`: A hyperparameter for your lle algorithm, as described above.

## 2 Jupyter Notebook (15 points)

Lastly, you will explore bias in word embeddings, and implement a debiasing method in the jupyter notebook we've provided to you called `FATE.ipynb`. Please complete the notebook by filling in all the TODOs in the coding cells, and answering the questions with red FILL IN text. You will submit your notebook to Gradescope.

## 3 Analytical (40 points)

**Question 1) (15 points) Expressive power of sequence classifiers** Consider the following sequence classification task. We are given an arbitrarily long sequence of tokens from the alphabet $\mathcal{X} = \{(, ), \texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}\}$. In other words, each sequence $\mathbf{x} \in \mathcal{X}^\star$ where $(\cdot)^\star$ denotes the Kleene closure operator. Our sequence classification task is to predict whether or not a given $\mathbf{x}$ is a well formed—meaning that the parentheses are balanced. Let $\mathcal{Y} = \{\texttt{Yes}, \texttt{No}\}$ denote the prediction.

*Examples*:

$x_1, \ldots, x_5 =$ `( ( ( ( (` $\Rightarrow y =$ `No`

$x_1, \ldots, x_3 =$ `) ( a` $\Rightarrow y =$ `No`

$x_1, \ldots, x_4 =$ `( ) ( )` $\Rightarrow y =$ `Yes`

$x_1, \ldots, x_8 =$ `( a ( b ) ( ) )` $\Rightarrow y =$ `Yes`

Notice that we are *not* predicting a $y$ label for each position, we are classifying the entire sequence with a $y$ label. However, the model for predicting the final classification will make use of hidden states at each position, $z_1 \ldots z_T$ where $T = |\mathbf{x}|$. Each latent state $z_t$ comes from a set $\mathcal{Z}$, the specific set will depend on the model family. For HMMs and CRFs, $\mathcal{Z}$ is a discrete set of symbols of some fixed size. For RNNs, $\mathcal{Z}$ is a real-valued vector of some fixed dimensionality. The latent state may be used to track the unobserved properties of the input sequence (e.g., some sort of latent representation of how the parentheses in the expression are matched).

When we are given a new sequence to classify $\mathbf{x}$, we infer the hidden $\mathbf{z}$ states according to our model parameters. We use the final state $z_T$, to predict whether or not the sequence was well formed. You can make this prediction by either examining the final state itself, or using the final state as input to a classifier that predicts `Yes` or `No`.

**1.1:** For each of the following family of sequence models, state as to whether there exists model parameters that can accurately capture the desired behavior. Explain.

(a) Hidden Markov Model

(b) Conditional Random Field

(c) Recurrent Neural Network

**1.2:** How would your answers to question 1.1 change if the sequences where upper bounded in length, $|\mathbf{x}| \leq T_{\max} < \infty$? Explain.

**Question 2) CRF vs. HMM (15 points)** Consider the following sequence models:

- HMM with $K = |\mathcal{Z}|$ latent states and $M = |\mathcal{X}|$ possible observations.
- Linear chain CRF over the same state space as the HMM, but with feature function $\boldsymbol{f}(\mathbf{x}, y_t, y_{t-1}) \in \mathbb{R}^D$.

For the following questions give a big-O expression that is a function of $K$, $D$, $M$, and input length $T$.

(a) Computing the posterior distribution over a specific label $p_{\text{HMM}}(Y_t = y \mid \mathbf{x})$

(b) Computing the posterior distribution over a specific label $p_{\text{CRF}}(Y_t = y \mid \mathbf{x})$

(c) How many parameters are in the HMM?

(d) How many parameters are in the CRF?

(e) List one advantage and one disadvantage of CRF over HMMs.

**Question 3) FATE (10 points)** Suppose you work for a bank and you have been tasked with building a machine learning system that automatically approves or rejects credit card applications. You are given a dataset that contains a sample of previously processed applications, for which a human decision maker decided to approve or reject. Each example contains information from the application, such as education level, zip code, and income level, as well as a label indicating the decision (accept/reject). The demographics of the applicant (gender, age, race, and ethnicity) are not included in the dataset or as features in the data.

(a) Despite the fact that demographic information is not included in the dataset, you are concerned that the resulting classifier may be biased against a demographic group. Name two potential sources of such bias.

(b) Take the same dataset, but add the observation of whether or not the applicant kept in good credit standing for 5 years after having their application approved (with rejected candidates labeled as "unknown, due to rejection"). Suppose we trained to predict the good credit standing rather than the person's judgement. How would that change the potential for bias? Explain.

(c) Would using an interpretable machine learning model necessarily reduce bias and increase fairness?

## 4 What to Submit

In this assignment you will submit two things.

1. **Submit your code (.py files) to cs475.org as a zip file**. **Your code must be uploaded as code.zip with your code in the root directory**. By 'in the root directory,' we mean that the zip should contain `*.py` at the root (`./*.py`) and not in any sort of substructure (for example `hw1/*.py`). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., `*.py`) rather than specifying a folder (e.g., `hw1`):

   ```
   zip code.zip *.py
   ```

   A common mistake is to use a program that automatically places your code in a subfolder. It is your job to make sure you have zipped your code correctly.

   We will run your code using the exact command lines described earlier, so make sure it works ahead of time, and make sure that it doesn't crash when you run it on the test data. A common mistake is to change the command line flags. If you do this, your code will not run.

   Remember to submit all of the source code, including what we have provided to you. We will include `requirements.txt` and provide the data, but nothing else.

2. **Submit your writeup to gradescope.com**. **Your writeup must be compiled from latex and uploaded as a PDF**. The writeup should contain all of the answers to the analytical questions asked in the assignment. Make sure to include your name in the writeup PDF and to use the provided latex template for your answers following the distributed template. You will submit this to the assignment called "Homework 5: Dimensionality Reduction, Structured Prediction, and FATE: Written".

3. **Submit your Jupyter notebook to gradescope.com**. Create a PDF version of your notebook (File $\rightarrow$ Download As $\rightarrow$ PDF via LaTeX (.pdf)). Be sure you have run the entire notebook so we can see all of your output. You will submit this to the assignment called "Homework 5: FATE: Notebook".

## 5 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/jkqbzabvyr15up`.