HOMEWORK 3

STAT640: DATA MINING AND STATISTICAL LEARNING
2015-2016, FALL SEMESTER
GENEVERA ALLEN

RICE UNIVERSITY

Michael Weylandt

2015-10-29

Problem 1 (Spam Data)

Please download the *spam data* from the course webpage to use for this problem.
K-Fold CV - Choose any classifier with one tuning parameter for parts (a-e)

(a) Write a function to perform K-fold cross-validation to select the tuning parameter for this classifier. You must code this up yourself and cannot use built-in functions (using a built-in function for the base classifier is fine).

(b) Select the optimal tuning parameter using
  (i) the minimum CV error rule; and
  (ii) the one SE rule
  for $K = 5$-fold CV. Are the models selected different? Interpret these results and reflect on this.

(c) Perform both $K = 5$ and $K = 10$ fold CV. Does this change the results? Is one of these preferable for this problem?

(d) When reporting the CV error, try out different loss functions:
  (i) misclassification error;
  (ii) binomial deviance error; and
  (iii) hinge loss error.
  Which error function is best for CV and model selection? Why?

(e) Reflect on your results. What have you learned about CV? Which approach to model selection do you think is best for this spam classification example? Why?

Process of Statistical Learning - Decide which classifier is best for building a spam filter.

(f) Use a model selection procedure to select tuning parameters for each of the following classifiers: Linear SVM, Gaussian Kernel SVM, and Polynomial Kernel SVM.

(g) Report the accuracy (model assessment) of each classifier for this spam data set. Which one is best? Why? Interpret and reflect on your results.

(h) Discuss why your model selection and assessment procedures are correct and justify any decisions you made.
  Note: For parts (f-h), you may use any built-in functions. The question is purposefully vague as it is up to you to design and implement a correct model selection and model assessment scheme to decide which type of SVM classifier is best for building a spam filter.

I use `Python` and `scikit-learn` ([PVG+11]) for my implementation. The underlying SVM implementations are due to `libsvm` ([CL11], for the general case) and `liblinear` ([FCH+08], for the linear case).

(a) If `perform_assessment=False`, I use $K-1$ folds for training and 1 for testing. If `perform_assessment=True`, I use $K-2$ folds for training, 1 for testing, and 1 for assessment.

```python
from __future__ import division

from collections import defaultdict

import numpy as np
import pandas as pd
import sklearn as sk
import sklearn.svm as svm
import matplotlib.pyplot as plt

import matplotlib.style
matplotlib.style.use('ggplot')

class CrossValidator(object):
    def __init__(self, classifier, K, param_name, param_min, param_max,
                       data, label_column="CLASS", perform_assessment=False,
                       loss_func=None, n_iter=None, param_steps=10,
                       *classifier_args, **classifier_kwargs):
        self.classifier = classifier
        self.K = K
        self.param_name = param_name
        self.param_min = param_min
        self.param_max = param_max
        self.param_steps = param_steps

        self.data = data.copy()

        if loss_func is None:
             self.loss_func = self.misclassification_err
        else:
             self.loss_func = loss_func

        self.n_iter = n_iter or K
        self.perform_assessment = perform_assessment

        self.classifier_args = classifier_args
        self.classifier_kwargs = classifier_kwargs

        self.label_column = label_column

        self._test_results = None
        self._assess_results = None
        self._test_err = None

    @property
    def param_range(self):
        return np.linspace(self.param_min,
                           self.param_max,
                           self.param_steps)

    def run(self):
        if self._test_results is not None:
            return

        self._test_results = defaultdict(list)
        self._assess_results = defaultdict(list)
        n_per_fold = int(self.data.shape[0] / self.K)
        assert n_per_fold > 0

        for param in self.param_range:
            for n in range(self.n_iter):
                ## select random test set for this iteration
                rand_ind = np.random.permutation(self.data.shape[0])
```

```
                TEST = self.data.iloc[rand_ind[:n_per_fold],:]
                TEST_DATA = sk.preprocessing.scale(TEST.drop(self.label_column, axis=1))
                TEST_LABELS = TEST[self.label_column]

                if self.perform_assessment:
                    ASSESS = self.data.iloc[rand_ind[n_per_fold:(2*n_per_fold)],:]
                    ASSESS_DATA = sk.preprocessing.scale(ASSESS.drop(self.label_column, axis=1))
                    ASSESS_LABELS = ASSESS[self.label_column]

                    TRAIN = self.data.iloc[rand_ind[2*n_per_fold:],:]
                    TRAIN_DATA = sk.preprocessing.scale(TRAIN.drop(self.label_column, axis=1))
                    TRAIN_LABELS = TRAIN[self.label_column]
                else:
                    TRAIN = self.data.iloc[rand_ind[n_per_fold:],:]
                    TRAIN_DATA = sk.preprocessing.scale(TRAIN.drop(self.label_column, axis=1))
                    TRAIN_LABELS = TRAIN[self.label_column]

                    ASSESS = ASSESS_DATA = ASSESS_LABELS = None

                ## run classifier
                kwargs = self.classifier_kwargs.copy()
                kwargs[self.param_name] = param

                classifier = self.classifier(*self.classifier_args, **kwargs)
                classifier.fit(TRAIN_DATA, TRAIN_LABELS)
                test_predict = classifier.decision_function(TEST_DATA)

                ## calculuate loss and record
                self._test_results[param].append(self.loss_func(predicted=test_predict,
                                                                 truth=TEST_LABELS))
                if self.perform_assessment:
                    assess_predict = classifier.decision_function(ASSESS_DATA)
                    self._assess_results[param].append(self.loss_func(predicted=assess_predict,
                                                                       truth=ASSESS_LABELS))

        self._test_err = {k:(sum(v)/len(v), np.std(v))
                          for k,v in self._test_results.items()}
        if self.perform_assessment:
            self._assess_err = {k:(sum(v)/len(v), np.std(v))
                                for k,v in self._assess_results.items()}

    @property
    def min_cv_err_param(self):
        self.run()
        ## Ugly, but works
        ## return parameter with lowest error
        return sorted(self._test_err.items(), key=lambda x: x[1])[0][0]

    @property
    def one_se_rule_param(self):
        self.run()
        ## Get 'optimal' param
        optim = sorted(self._test_err.items(), key=lambda x: x[1])[0][0]
        optim_err, optim_err_se = self._test_err[optim]

        ## Now 'simplify' to a lower value of param
        for par, (cv_err, cv_err_se) in sorted(self._test_err.items()):
            if cv_err <= optim_err + optim_err_se:
                return par

    ## Some built in loss functions for part (d)
    @staticmethod
    def misclassification_err(predicted, truth):
```

```
            return (np.sign(predicted) != truth).mean()

        @staticmethod
        def hinge_loss_err(predicted, truth):
            return np.mean(np.maximum(0, 1 - predicted * truth))

        @staticmethod
        def binomial_deviance_err(predicted, truth):
            return np.mean(np.log(1 + np.exp(-2*predicted * truth)))

        def plot(self, title=None, filename=None):
            plt.figure()

            plt.errorbar(self._test_err.keys(),
                         [x[0] for x in self._test_err.values()],
                         [x[1] for x in self._test_err.values()],
                         fmt="o", ecolor="g", capthick=2,
                         label="Test Error + SE of Estimate");

            if title is None:
                title="Parameter (%s) Selection: $K=%s$-Fold CV" % (self.param_name, self.K)

            plt.title(title)
            plt.xlabel("Parameter (%s)" % self.param_name)
            plt.ylabel("Test Error")

            plt.legend(loc="best", frameon=True, prop={"size":"x-small"})

            if filename:
                plt.savefig(filename)
            else:
                plt.show()

        def predicted_test_err(self, param):
            self.run()
            return self._assess_err[param]
```

(b) We use the spam data set and the ν-SVM characterization ([SSWB00]) as it provides a more intuitive tuning parameter.[1] We do not perform model assessment in the first part of this question.

```
feature_names = []
with open("spam_vars.txt") as f:
    for l in f:
        feature_names.append(l.split(":")[0])

feature_names.append("CLASS")

SPAM = pd.read_csv("spam_dat.csv", header=None, names=feature_names)
SPAM.loc[SPAM["CLASS"]==0, "CLASS"] = -1 ## Conventional for SVM

cv5 = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                     param_min=0.1, param_max=0.7, data=SPAM,
                     kernel="linear")
cv5.run()

cv10 = CrossValidator(classifier=svm.NuSVC, K=10, param_name="nu",
                      param_min=0.1, param_max=0.7, data=SPAM,
                      kernel="linear")
cv10.run()
```

We find that, in this case, 5-fold cross validation gives slightly different results for the two decision rules. The actual truth may lie somewhere between these two as we used a fairly coarse ν-grid.

[1] Note that, since ν is a *lower bound* on the number of support vectors (*cf.* Proposition 1(ii) of [SSWB00]), decreasing ν regularizes the model. (Lower the lower bound $\Longrightarrow$ fewer support vectors $\Longrightarrow$ more regularization)

```
cv5.min_cv_err_param
```

0.30000000000000004

and

```
cv5.one_se_rule_param
```

0.23333333333333334

(c) When we add 10-fold CV we note that we get one ν for both rules – this is likely because the error bars used in the one-standard-error rule are smaller when we have less data 'held out' in each iteration of CV.

```
cv10.min_cv_err_param
```

0.23333333333333334

and

```
cv10.one_se_rule_param
```

0.23333333333333334

(d) We can fit other loss functions, here we focus a narrower range of ν based on the results above:[2]

```
cv5_hl = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                        param_min=0.1, param_max=0.3, data=SPAM,
                        kernel="linear",
                        loss_func=CrossValidator.hinge_loss_err)
cv5_hl.run()

cv5_bd = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                        param_min=0.1, param_max=0.3, data=SPAM,
                        kernel="linear",
                        loss_func=CrossValidator.binomial_deviance_err)
cv5_bd.run()
```

Using these loss functions (and narrower ν), we find that our fitted parameter changed from above:

```
cv5_hl.min_cv_err_param
```

0.21111111111111111

and

```
cv5_hl.one_se_rule_param
```

0.21111111111111111

For binomial deviance,

```
cv5_bd.min_cv_err_param
```

0.23333333333333331

and

```
cv5_bd.one_se_rule_param
```

0.21111111111111111

We note here that the binomial deviance loss is the only loss function which leads to a different $\hat{\nu}$ under the two decision rules.

I would prefer the misclassification error here as it's more intuitive, but use of a hinge loss or binomial deviance likely have better statistical properties (since HL implicitly rewards a large margin and BD is MLE). If misclassification is used, it may be better to use a weighted misclassification which penalizes false positives more highly than false negatives; the cost of a missed email is typi-

[2] This is not *kosher* as a model selection step, but it is useful for comparing different forms of cross validation since we used a very coarse grid for ν in previous sections.

cally much higher than the cost of having to delete an email.

(e) The variety of answers above indicate that selecting parameters by CV is more of an art than a science. We can get different (but not radically so) 'optimal' ν by varying our loss function, our decision rule, or K.

Cross Validation works well here, but the data set is large enough that a data splitting approach could also work well.

(f) We continue like before, but here we use different kernels. Since our `CrossValidator` is only set up to search over one parameter (ν) we do not attempt to tune our kernels and use the default parameters.[3] Here we do want to perform model assessment and we modify our call to `CrossValidator` accordingly:

```
cv5_linear = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                            param_min=0.1, param_max=0.3, data=SPAM,
                            kernel="linear", perform_assessment=True)

cv5_linear.run()

cv5_gaussian = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                              param_min=0.1, param_max=0.3, data=SPAM,
                              kernel="rbf", perform_assessment=True)  ## Radial basis = Gaussian

cv5_gaussian.run()

cv5_poly = CrossValidator(classifier=svm.NuSVC, K=5, param_name="nu",
                          param_min=0.1, param_max=0.3, data=SPAM,
                          kernel="poly", perform_assessment=True)

cv5_poly.run()
```

For these kernels, we find a different value of ν appears optimal than for the linear case; this is not surprising, with a more flexible decision boundary, we need more regularization.

For the linear kernel:

```
cv5_linear.min_cv_err_param
```

```
0.21111111111111111
```

and

```
cv5_linear.one_se_rule_param
```

```
0.18888888888888888
```

For the Gaussian kernel:

```
cv5_gaussian.min_cv_err_param
```

```
0.16666666666666666
```

and

```
cv5_gaussian.one_se_rule_param
```

```
0.14444444444444443
```

For the Polynomial kernel:

```
cv5_poly.min_cv_err_param
```

[3] For the Gaussian (radial basis) kernel, the default kernel is

$$K(x, y) = \exp\left\{-\frac{|\|x-y\|_2^2}{p}\right\}$$

For the polynomial kernel, the default kernel is

$$K(x, y) = \frac{1}{p^3}\langle x, y\rangle^3$$

```
0.18888888888888888
```

and

```
cv5_poly.one_se_rule_param
```

```
0.10000000000000001
```

(g) Since our cross validation was done with an assessment set, we can recover estimated test error (misclassification error) directly.[4] For the linear kernel:

```
cv5_linear.predicted_test_err(cv5_linear.one_se_rule_param)
```

```
(0.080217391304347824, 0.011735103979361549)
```

and

```
cv5_gaussian.predicted_test_err(cv5_gaussian.one_se_rule_param)
```

```
(0.066956521739130442, 0.0059376088190207215)
```

and

```
cv5_poly.predicted_test_err(cv5_gaussian.one_se_rule_param)
```

```
(0.078043478260869562, 0.0070241280093058739)
```

The first number is the predicted test error – the second is the estimated standard deviation (not standard error) of the test error.

Based on the above, the Gaussian kernel appears to perform the best (lowest error on the training set).

(h) The analysis above doesn't perform any tuning on the kernel parameters and this is a serious limitation. The test error above is still unacceptably high and the kernel must be tuned to get acceptable performance.

Considering only tuning ν: I chose to 're-fold' our data for each CV iteration, rather than simply using each fold as a test set once. This doesn't guarantee us as much mixing as standard CV, but it does allow us to use $n > K$ iterations for each K which I think is a win.

Since this data set is relatively large ($n = 4601$), we probably could have used a simpler (data splitting) approach rather than cross-validation.

[4]For ease of writing code, we calculated the predicted test error for all values of ν and trust the user not to look. In practice, we should probably not calculate until requested since its hard to resist the temptation to look.

Problem 2 (Handwritten digits)

Use PCA, NMF, and ICA to find patterns, reduce the dimension, and visualize the data. Please download the *Digits Data* from the ESL webpage.

(a) Visualize results from the 3 methods. How would you visualize patterns among the samples? Among the features? Show these graphics, explain them, and interpret the results. What do these reveal? Do you find anything interesting?

(b) How much variance is explained by each PC? What would be a good number of PC factors to retain for this data? Explain.

(c) How do the results of ICA and NMF change when you take $r = 10, 20, 50, 250$ factors? Is there a way that you could decide how many factors to retain in a data-driven manner? Explain.

(d) Is there a quantitative and objective way to that you can determine which is the best pattern recognition technique for this data set? How? Explain and implement your procedure.

As done in class, we restrict our attention to the 3's in the data set; we use the test set data only (since we are not making predictions).

```
import gzip
with gzip.open("zip.test.gz") as f:
    test_set = pd.read_csv(f, sep=" ", header=None)

test_threes = test_set[test_set[0]==3].drop(0, axis=1)

from sklearn.decomposition import PCA, NMF
from sklearn.decomposition import FastICA as ICA
```

(a) PCA returns the most 'evident' threes: in this case (looking only at one digit), there's really only one main pattern (the 3 shape) with variants so the NMF- and ICA-identified independent sources are as easy to see. (They look more like parts of a 3 than a different 'style' of a 3) The major patterns of PCA, after the first, are 'heaviness' (indicative of weightier penmanship) and 'wideness' (compared to the 1st PC which is sort of skinny).

For each method, we can visualize columns as either 'average observations' ($n \times k$ matrix) and 'feature patterns' ($k \times p$ matrix).

(b) Figure 2.1 is a graph of the variance explained by each principal component (a screeplot):

```
_, D, _ = np.linalg.svd(test_threes)
eigenvalues = D**2
n_pcs = 20
fig, axes = plt.subplots(nrows=2, ncols=1)
ax1, ax2 = axes.flat

ax1.plot(eigenvalues[:n_pcs]/np.sum(eigenvalues))
ax1.set_xlabel("Principal Component")
ax1.set_ylabel("Additional Variance Explained")

ax2.plot(np.cumsum(eigenvalues)[:n_pcs]/np.sum(eigenvalues))
ax2.set_xlabel("Principal Component")
ax2.set_ylabel("Total Variance Explained")
```

From the graph, it's not clear what an acceptable number of principal components would be. The 'elbow' rule suggests either 1 or 5 PCs, but neither of these explain a large fraction of the variance.

For a more rigorous method to compare models (both choosing the number of principal and comparing different model classes) see part (d).

(c) We create an analogue to Figure 2.1 for both NMF and ICA. Instead of variance, however, we compare the Frobenius norm of the difference between the original and the approximating matrix.

For NMF, see Figure 2.2:

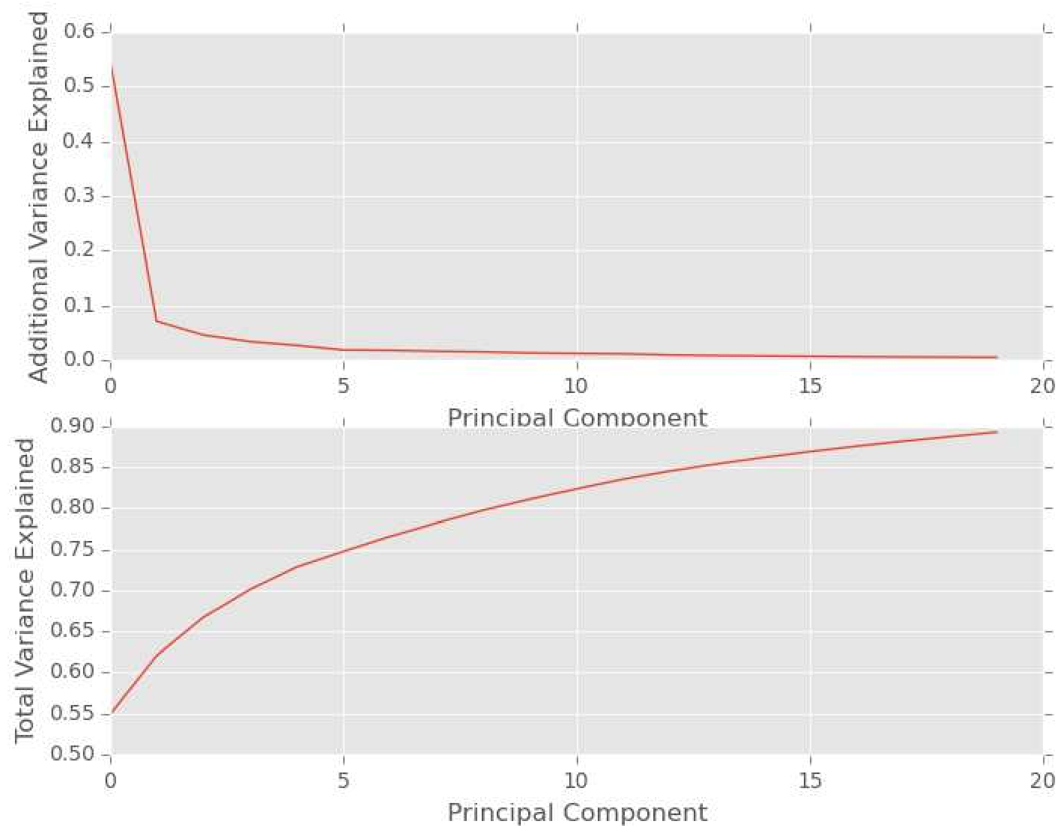

Figure 2.1: Screeplot for 2(b)

```
test_threes_nn = test_threes - np.min(test_threes) ## Make non-negative

scree_nmf = {}

for r in (10, 20, 50, 150, 250):
    for _ in range(3):
        n = NMF(n_components=r)
        W = n.fit_transform(test_threes_nn.T)
        H = n.components_

        scree_nmf[r] = np.linalg.norm(test_threes_nn.T - np.dot(W, H))/3

plt.plot(scree_nmf.keys(),
         scree_nmf.values(), "ro")
plt.ylabel("Frobenius Norm of Reconstruction Error")
plt.xlabel("Rank of NMF approximation")
plt.title("Scree-ish Plot: NMF (Problem 2(c))")
```

We note that, as expected, the reconstruction error is decreasing in the number of components used. The large rise at $r = 250$ is surprising, but likely an artifact of the fact we are using only 166 training images.

Similarly for ICA (omitting $r = 250$ because ICA can't be performed in that case with our small data set):

```
scree_ica = {}

for r in (10, 20, 50, 150):
    for _ in range(3):
```

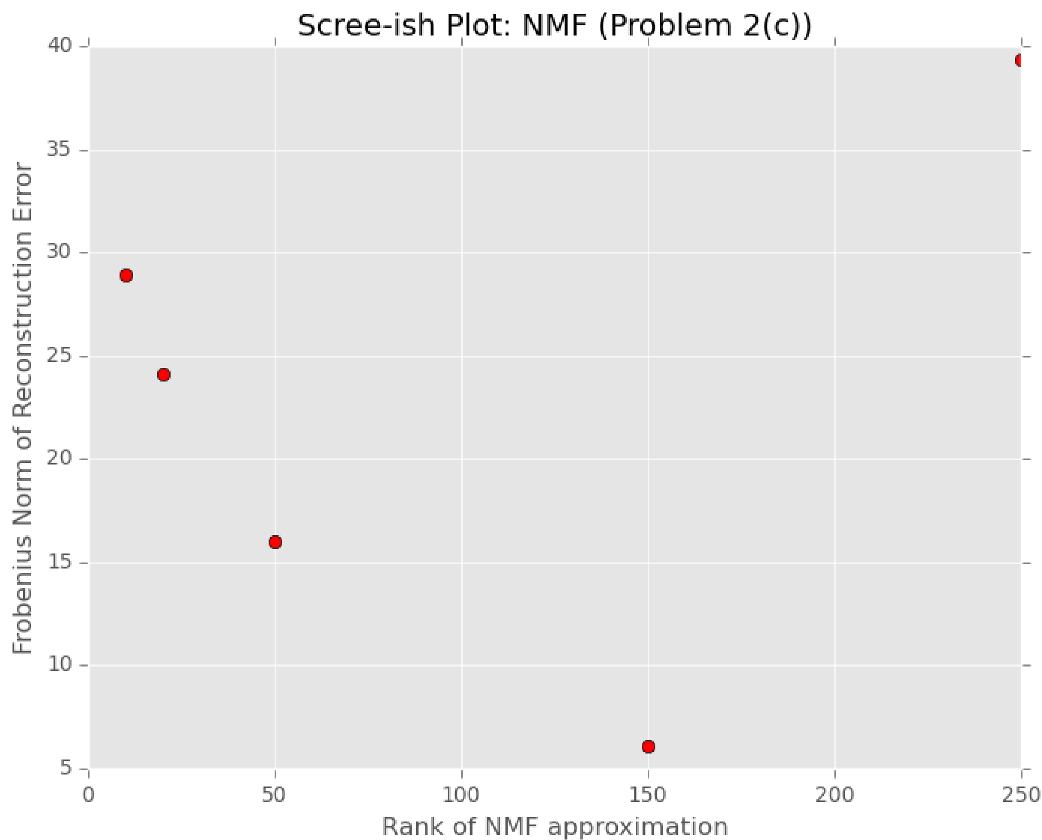

Figure 2.2: NMF Scree-like plot for 2(b)

```
        n = ICA(n_components=r)
        W = n.fit_transform(test_threes)
        H = n.mixing_.T

        scree_ica[r] = np.linalg.norm(test_threes - np.dot(W, H))/3

plt.plot(scree_ica.keys(),
         scree_ica.values(), "ro")
plt.ylabel("Frobenius Norm of Reconstruction Error")
plt.xlabel("Rank of ICA approximation")
plt.title("Scree-ish Plot: ICA (Problem 2(c))")
```

As before, we see the reconstruction error increasing with r.

For a more rigorous method to compare models (both choosing r and comparing different model classes) see part (d).

(d) One method we could use to compare matrix factorization methods would be to compare their effectiveness at imputation, effectively transforming them into a prediction problem by 'masking' some of the data and seeing how well each model reconstructs the original data given only the masked data.[5] We implement this technique for PCA below:

```
from scipy.stats import nanmean

def pca_impute(data, num_pc=10, pct_missing=0.02, tol=0.0001, n_iter=5):
   TOTAL_ERR = 0
   ORIGINAL_DATA = np.array(data).copy()
   for _ in range(n_iter):
```

[5] This method is discribed in [HTS+99]. The algorithm here is described in Section 1.2.

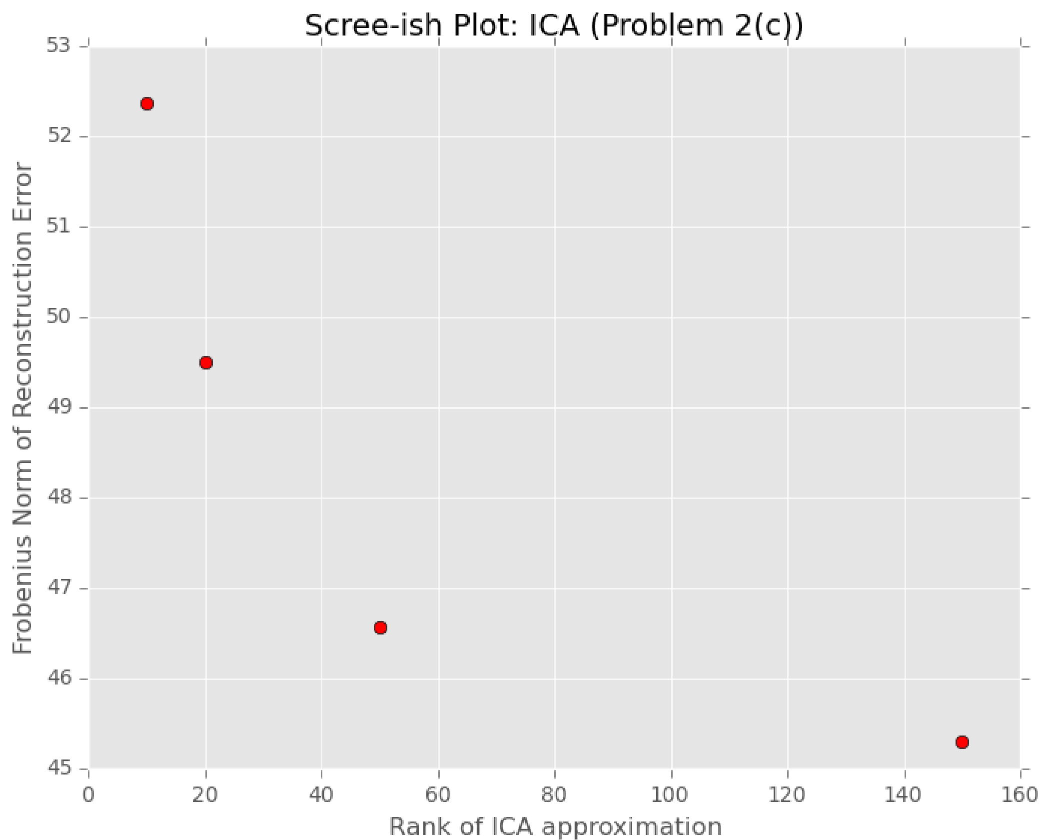

Figure 2.3: ICA Scree-like plot for 2(b)

```
        ## We make a copy of our data and NaN-out
        ## (mark as missing) some points at random
        impute_data = ORIGINAL_DATA.copy()
        num_missing = int(pct_missing * impute_data.size)

        masking_ind = np.unravel_index(np.random.permuatation(impute_data.size)[:num_missing],
                                       impute_data.shape)
        ## randomly mask some data
        impute_data[masking_ind] = np.nan

        this_iter = impute_data.copy()
        ## Initialize with a bad 'last_iter'
        ## so we don't accidentally abort early
        last_iter = np.zeros_like(impute_date)

        while np.linalg.norm(this_iter - last_iter)/np.lingalg.norm(this_iter) > tol:
            last_iter = this_iter
            last_U, last_D, last_V = np.lingalg.svd(last_iter)

            ## Scipy broadcasting takes care of nanmean(impute_data) being px1 here
            this_iter[masking_ind] = (nanmean(impute_data,axis=1) + np.dot(last_U, last_D, last_V))[masking_ind]

         TOTAL_ERR = np.lingalg.norm(this_iter - ORIGINAL_DATA)

    return TOTAL_ERR/n_iter
```

We mask certain points randomly and then use a SVD based approach to impute those data with a regression like technique (note that here we use column means as an intercept for our 'regression' because PCA only picks up variance). We then take the SVD of our imputed data and use it to

re-impute until convergence.

Problem 3 (Non-Negative Matrix Factorization)

Derive an algorithm to find a local optimum of the non-negative matrix factorization problem commonly employed for count valued data:

$$\begin{aligned} \underset{\mathbf{W},\mathbf{H}}{\text{maximize}} \quad & \sum_{i=1}^{n}\sum_{j=1}^{p}\left(X_{ij}\log\left(\sum_{k=1}^{K}W_{ik}H_{kj}\right) - \sum_{k=1}^{K}W_{ik}H_{kj}\right) \\ \text{subject to} \quad & W_{ik} \geq 0 \quad \forall i = 1,\dots,n; k = 1,\dots,K \\ & H_{kj} \geq 0 \quad \forall j = 1,\dots,p; k = 1,\dots K \end{aligned}$$

Hint: There are several possible algorithmic strategies that can be used. If you employ an MM (majorize-minimize) algorithm, you may find ESL problem 14.23 helpful.

As hinted, we use a MM algorithm as described in Problem 14.23 of [HTF09].[6]

We begin by noting that, because $\log(\cdot)$ is a concave function, the following holds:[7] for any convex combination of $\{y_i\}_{i=1}^k$ with weights $\{c_i\}_{i=1}^k$ ($c_i \in [0,1], \forall i; \sum_{i=1}^k c_i = 1$),

$$\begin{aligned} \log\left(\sum_{i=1}^{k} y_i\right) &= \log\left(\sum_{i=1}^{k} c_i * y_i/c_i\right) \\ &\geq \sum_{i=1}^{k} c_i \log(y_i/c_i) \end{aligned}$$

We can then apply this to find that:

$$\log\left(\sum_{k=1}^{K} w_{ik}h_{kj}\right) \geq \sum_{k=1}^{K} \frac{a_{ikj}^s}{b_{ij}^s}\log\left(\frac{b_{ij}^s}{a_{ikj}^s}w_{ik}h_{kj}\right)$$

where

$$\begin{aligned} a_{ikj}^s &= w_{ik}^s h_{kj}^s \\ b_{ij}^s &= \sum_{k=1}^{K} a_{ikj}^s \\ &= \sum_{k=1}^{K} w_{ik}^s h_{kj}^s \end{aligned}$$

[6] This problem derives a MM algorithm (*cf.* [WL10], unpublished when [HTF09] was published and referred to as unpublished in the bibliography) for NMF.

[7] A function ϕ is *concave* if $-\phi$ is convex. If ϕ is concave, then:

$$\begin{aligned} -\phi\left(\lambda x + (1-\lambda)y\right) &\leq \lambda * -\phi(x) + (1-\lambda) * -\phi(y) \qquad \text{(By convexity of } -\phi\text{)} \\ \phi\left(\lambda x + (1-\lambda)y\right) &\geq \lambda\phi(x) + (1-\lambda)\phi(y) \end{aligned}$$

This extends to arbitrary convex combinations of points by induction: let $c_1,\dots,c_k \in [0,1]$ with $\sum_i c_i = 1$: then,

$$\begin{aligned} \phi\left(\sum_{i=1}^{k} c_i x_i\right) &= \phi\left(\overbrace{c_1}^{=\lambda} x_1 + \overbrace{\sum_{i=2}^{k} c_k x_k}^{=(1-\lambda)\overline{y}}\right) \\ &\geq c_1\phi(x_1) + \phi\left(\sum_{i=2}^{k} c_k x_k\right) \\ &\geq \sum_{i=1}^{k} c_i \phi(x_i) \qquad \text{inductively} \end{aligned}$$

Since $w_{ik}, h_{kj} \geq 0$, we see that $0 \leq a^s_{ikj}$ and $\frac{a^s_{ikj}}{b_{ij}}$ are the coefficients of a convex combination for each (i, j). (We can ignore s here – we will use it for an iteration counter later)

Now consider two functions:

$$L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{i=1}^n \sum_{j=1}^p \left(X_{ij} \log \left(\sum_{k=1}^K w_{ik} h_{kj} \right) - \sum_{k=1}^K w_{ik} h_{kj} \right)$$
$$g(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}^s, \boldsymbol{H}^s) = \sum_{i=1}^n \sum_{j=1}^p \sum_{k=1}^K u_{ij} \frac{a^s_{ikj}}{b^s_{ij}} (\log w_{ik} + \log h_{kj}) - w_{ik} h_{kj}$$

fro some $\boldsymbol{U}$ depending on $\boldsymbol{X}$.

We note that g minorizes L.[8] It is easier to look at L_{ij}, g_{ij} which are the summands (fix i, j in the outer sums):

$$L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = \left(X_{ij} \log \left(\sum_{k=1}^K w_{ik} h_{kj} \right) - \sum_{k=1}^K w_{ik} h_{kj} \right)$$
$$g_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}^s, \boldsymbol{H}^s) = \sum_{k=1}^K u_{ij} \frac{a^s_{ikj}}{b^s_{ij}} (\log w_{ik} + \log h_{kj}) - w_{ik} h_{kj}$$

We first show that $g_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}^s, \boldsymbol{H}^s) \leq L_{ij}(\boldsymbol{W}, \boldsymbol{H})$:

$$\begin{aligned}
g_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}^s, \boldsymbol{H}^s) &= \sum_{k=1}^K u_{ij} \frac{a^s_{ikj}}{b^s_{ij}} (\log w_{ik} + \log h_{kj}) - w_{ik} h_{kj} \\
&= u_{ij} \left(\frac{a^s_{ikj} b_{ij}}{\sum} \,_{k=1}^{k} w_{ik} h_{kj} \log(w_{ik} h_{kj}) \right) - \sum_{k=1}^K w_{ik} h_{kj} \\
&\leq u_{ij} \left(\frac{a^s_{ikj} b_{ij}}{\sum} \,_{k=1}^{k} w_{ik} h_{kj} \log(\frac{b_{ij}}{a^s_{ikj}} w_{ik} h_{kj}) \right) - \sum_{k=1}^K w_{ik} h_{kj} \\
&\leq u_{ij} \log \left(\sum_{k=1}^k w_{ik} h_{kj} \right) - \sum_{k=1}^K w_{ik} h_{kj} \\
&\leq L_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}^s, \boldsymbol{H}^s)
\end{aligned}$$

if $u_{ij} < X_{ij}$.

We next show that $g_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}, \boldsymbol{H}) = L_{ij}(\boldsymbol{W}, \boldsymbol{H})$:

$$\begin{aligned}
g_{ij}(\boldsymbol{W}, \boldsymbol{H} | \boldsymbol{W}, \boldsymbol{H}) &= \sum_{k=1}^K u_{ij} \frac{a_{ikj}}{b_{ij}} (\log w_{ik} + \log h_{kj}) - w_{ik} h_{kj} \\
&= \frac{u_{ij}}{b_{ij}} \left(\sum_{k=1}^K w_{ik} h_{kj} \log(w_{ik} h_{kj}) \right) - \sum_{k=1}^K w_{ik} h_{kj} \\
&= L_{ij}(\boldsymbol{W}, \boldsymbol{H})
\end{aligned}$$

when

$$\begin{aligned}
X_{ij} \log \left(\sum_{k=1}^K w_{ik} h_{kj} \right) &= \frac{u_{ij}}{b_{ij}} \left(\sum_{k=1}^K w_{ik} h_{kj} \log(w_{ij} h_{kj}) \right) \\
\implies u_{ij} &= \frac{X_{ij} b_{ij} \log \left(\sum_{k=1}^k w_{ik} h_{kj} \right)}{\left(\sum_{k=1}^K w_{ik} h_{kj} \log(w_{ij} h_{kj}) \right)} \\
&= X_{ij} \frac{\left(\sum_{k=1}^K w_{ik} h_{kj} \right) \log \left(\sum_{k=1}^K w_{ik} h_{kj} \right)}{\left(\sum_{k=1}^K w_{ik} h_{kj} \log(w_{ij} h_{kj}) \right)}
\end{aligned}$$

[8] Recall that a function $g(x, y)$ minorizes $f(x)$ if $g(x, x) = f(x)$ and $g(x, y) \leq f(x), \forall x, y$. Here $x = (\boldsymbol{W}, \boldsymbol{H})$ and $y = (\boldsymbol{W}^s, \boldsymbol{H}^s)$.

which is consistent with our requirement $u_{ij} < X_{ij}$ from above since $x\log(x)$ is a convex function.

We can then use an MM algorithm to derive updating steps for $\boldsymbol{W}^s, \boldsymbol{H}^s$.[9] Since g is continuous in $\boldsymbol{W}^s, \boldsymbol{H}^s$, maximization of g is direct:[10]

$$\begin{aligned}
\frac{\partial g}{\partial w_{ik}^s} &= \sum_{i=1}^n \sum_{j=1}^p \sum_{k=1}^K u_{ij} \log(w_{ik} h_{kj}) \frac{\partial}{\partial w_{ik}^s} \left(\frac{a_{ikj}^s}{b_{ij}^s} \right) \\
&= \sum_{i=1}^n \sum_{j=1}^p \sum_{k=1}^K u_{ij} h_{kj}^s * \frac{b_{ij}^s - a_{ikj}^s}{(b_{ij}^s)^2}
\end{aligned}$$

By setting this equal to zero and isolating w_{ik}^s, we can derive our update step:

$$\begin{aligned}
0 &= \frac{\partial g_{ik}}{\partial w_{ik}^s} \\
&= \sum_{j=1}^p u_{ij} h_{kj}^s * \frac{b_{ij}^s - a_{ikj}^s}{(b_{ij}^s)^2} \\
\sum_{j=1}^p \frac{u_{ij} h_{kj}^s}{b_{ij}^s} &= \sum_{j=1}^p \frac{a_{ikj}^s}{(b_{ij}^s)^2} \\
&= \sum_{j=1}^p \frac{w_{ik}^s h_{kj}^s}{(b_{ij}^s)^2} \\
\sum_{j=1}^p u_{ij} h_{kj}^s &= \sum_{j=1}^p \frac{w_{ik}^s h_{kj}^s}{b_{ij}^s} \\
w_{ik}^s &= \frac{\sum_{j=1}^p u_{ij} h_{kj}^s b_{ij}^s}{\sum_{j=1}^p h_{kj}^s} \\
&= \frac{\sum_{j=1}^p u_{ij} h_{kj}^s b_{ij}^s}{\sum_{j=1}^p h_{kj}^s} \\
&= w_{ik}^s \frac{\sum_{j=1}^p h_{kj} x_{ij} / b_{ij}^s}{\sum_{j=1}^p h_{kj}}
\end{aligned}$$

giving update step: (using the relationship between u, x found above)

$$w_{ik}^{s+1} \to w_{ik}^s \frac{\sum_{j=1}^p h_{kj} x_{ij} / (\boldsymbol{W}^s \boldsymbol{H}^s)_{ij}}{\sum_{j=1}^p h_{kj}}$$

A parallel derivation gives

$$h_{kj}^{s+1} \to h_{kj}^s \frac{\sum_{i=1}^n w_{ik} x_{ij} / (\boldsymbol{W}^s \boldsymbol{H}^s)_{ij}}{\sum_{i=1}^n w_{ik}}$$

[9] In general, MM algorithms provide updating steps of the form

$$x^{s+1} = \arg\max_x g(x, x^s)$$

See Chapter 6 of [Lan04] for details.

[10] Recall that in general #+BEGIN LaTeX

$$\begin{aligned}
\frac{\partial}{\partial x} \frac{ax}{ax+by+cz} &= \frac{(ax+by+cz) * a - ax * a}{(ax+by+cz)^2} \\
&= \frac{a^2x + aby + acz - a^2 x}{(ax+by+cz)^2} \\
&= \frac{aby + acz}{(ax+by+cz)^2} \\
&= a\frac{(ax+by+cz) - ax}{(ax+by+cz)^2}
\end{aligned}$$

Putting these together we find a workable algorithm:

$$w_{ik}^{s+1} \to w_{ik}^{s} \frac{\sum_{j=1}^{p} h_{kj} x_{ij} / (\boldsymbol{W}^s \boldsymbol{H}^s)_{ij}}{\sum_{j=1}^{p} h_{kj}}$$
$$h_{kj}^{s+1} \to h_{kj}^{s} \frac{\sum_{i=1}^{n} w_{ik} x_{ij} / (\boldsymbol{W}^s \boldsymbol{H}^s)_{ij}}{\sum_{i=1}^{n} w_{ik}}$$

(*Cf.* Equation 14.17 in [HTF09])

References

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[FCH+08] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. http://www.jmlr.org/papers/volume9/fan08a/fan08a.pdf.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2nd edition, February 2009. http://statweb.stanford.edu/~tibs/ElemStatLearn/.

[HTS+99] Trevor Hastie, Robert Tibshirani, Gavin Sherlock, Michael Eisen, Patrick Brown, and David Bostein. Imputing missing data for gene expression arrays, 1999. http://web.stanford.edu/~hastie/Papers/missing.pdf.

[Lan04] Kenneth Lange. *Optimization*. Springer Texts in Statistics. Springer, 2004.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf; http://scikit-learn.org.

[SSWB00] Bernhard Schlkopf, Alex J. Smola, Robert C. Williamson, and Peter L. Bartlett. New support vector algorithms. *Neural Computation*, 12:1207–1245, 2000. http://www.mitpressjournals.org/doi/pdf/10.1162/089976600300015565.

[WL10] Tong Tong Wu and Kenneth Lange. The MM alternative to EM. *Statistical Science*, 25:492–505, 2010. http://projecteuclid.org/download/pdfview_1/euclid.ss/1300108233.