MAT 170 Homework Project 3

Joe Lenning Student id: 919484830

May 19, 2024

1 Problem 3A

1.1 Code for Matrix

```
import csv
import datetime
import numpy as np
import os

def read_values_hist(RAFI_combined):
    with open('RAFI_combined', 'r') as my_file:
        reader = csv.DictReader(RAFI_combined)
        mydict = {}
        for row in reader:
            # Convert the date format from the CSV and capture the values
            date = datetime.datetime.strptime(row['Date'], '%m/%d/%Y')
            mydict[date] = float(row['Value_With_Dividends__USD_'])
    return mydict

def next_month(d):
    return datetime.datetime(d.year + (d.month // 12), ((d.month % 12) + 1), 1)

def find_closest_date(data, target_date):
    """Find the closest date within four days before or after the 1st of the month."""
    for offset in range(-4, 5):  # From four days before to four days after
        close_date = target_date + datetime.timedelta(days=offset)
        if close_date in data:
            return close_date
    return None
```

```python

# Paths to the data files
file_paths = {
    "RAFI Global": 'path/to/RAFI Global.csv',
    "RAFI US": 'path/to/RAFI US.csv',
    "RAFI EU": 'path/to/RAFI EU.csv',
    "RAFI Emerging Markets": 'path/to/RAFI Emerging Markets.csv',
    "RAFI Developed ex-US": 'path/to/RAFI Developed US.csv'
}

#data from files
index_data = {name: read_values_hist(path) for name, path in file_paths.items()}

#dates
start_date = datetime.datetime(1996, 7, 1)
end_date = datetime.datetime(2023, 12, 1)

all_dates = [start_date + datetime.timedelta(days=(next_month(start_date) - start_date).days * i) for i in range

returns_matrix = np.zeros((len(file_paths), len(all_dates) - 1))  # 5 indices and 329 months

for i, (name, data) in enumerate(index_data.items()):
    for j in range(1, len(all_dates)):
        prev_date = find_closest_date(data, all_dates[j-1])
        curr_date = find_closest_date(data, all_dates[j])
        if prev_date and curr_date:
            # Calculate return as (V(t) / V(t_prev)) - 1
            returns_matrix[i, j-1] = (data[curr_date] / data[prev_date]) - 1

# Center the matrix by subtracting the mean of each row
returns_centered = returns_matrix - np.mean(returns_matrix, axis=1, keepdims=True)

# Outputs the full 5x329 matrix centered by the mean
returns_centered
```

1.2 PCA Model

First we compute the covariance matrix Σ of the data X:

$$\Sigma = \frac{1}{n-1} X^T X$$

The first principal component vector v_1 is obtained by solving the following optimization problem:

$$\max_v v^T \Sigma v \quad \text{subject to} \quad \|v\| = 1$$

This optimization problem maximizes the variance of the projected data and is subject to the constraint that v has unit length.

The principal vector v_1 corresponds to the eigenvector of Σ associated with the largest eigenvalue. This can be derived from the eigenvalue equation:

$$\Sigma v = \lambda v$$

where λ is an eigenvalue of Σ and v is the corresponding eigenvector.

1.3 U Σ for Data

The following Python code uses the **numpy** library to perform Singular Value Decomposition (SVD) and extracts the matrices U, Σ, and V^T:

```python
import numpy as np

# Assuming 'returns_centered' is the centered matrix from your data processing
U, Sigma, VT = np.linalg.svd(returns_centered, full_matrices=False)
```

The matrix U and diagonal values from Σ are obtained as outputs from the SVD operation.

$$U = \begin{bmatrix} -0.4013 & -0.2745 & 0.2612 & 0.1065 & -0.8271 \\ -0.3409 & -0.4085 & 0.4009 & -0.6626 & 0.3423 \\ -0.4894 & -0.1078 & -0.8417 & -0.2005 & -0.0184 \\ -0.5499 & 0.8005 & 0.2252 & -0.0397 & 0.0672 \\ -0.4253 & -0.3245 & 0.1095 & 0.7126 & 0.4404 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

$$\Sigma = \begin{pmatrix} 1.2009 & 0 & 0 & 0 & 0 \\ 0 & 0.4053 & 0 & 0 & 0 \\ 0 & 0 & 0.3047 & 0 & 0 \\ 0 & 0 & 0 & 0.2097 & 0 \\ 0 & 0 & 0 & 0 & 0.0173 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

1.4 Error Term

Again we can just add the following snippet to our code

```python
total_variance = np.sum(sigma**2)
eta_k = (sigma**2) / total_variance

# Output the eta_k ratios
print("Ratios _k:", eta_k)
```

We see the result as The ratios of explained variance for each principal component derived from the singular values are:

$$\begin{aligned}
\eta_1 &\approx 0.8271 \quad (82.71\%) \\
\eta_2 &\approx 0.0942 \quad (9.42\%) \\
\eta_3 &\approx 0.0532 \quad (5.32\%) \\
\eta_4 &\approx 0.0252 \quad (2.52\%) \\
\eta_5 &\approx 0.0002 \quad (0.02\%)
\end{aligned}$$

These values indicate how much variance l component takes from the total variance of the data set. A high η_k suggests that a significant portion of the data's variation can be observed and analyzed effectively when projected onto the corresponding k-dimensional subspace(per the book). This insight is particularly useful for dimensionality reduction, as it helps to identify how many components should be retained to effectively capture the main characteristics while reducing rank.

2 Problem 3B, Least squares via pseudoinverse

2.1 a

The set of all optimal solutions to the least squares problem $\min \|Ax - b\|^2$ can be denoted as χ, where:

$$\chi = \{x \mid x = x^* + v, \text{ for all } v \in \text{Null}(A)\}$$

Here, x^* is any particular solution to the equation $Ax = \text{Proj}_{\text{Range}(A^T)}(b)$ and $\text{Null}(A)$ represents the null space of A.

2.2 b

Let $A \in R^{m \times n}$ and assume A has full rank. The solution to the least squares problem, $x^* = A^+ b$, can be shown to be optimal by the following derivation:

First, consider the least squares problem formulated as minimizing $\|Ax - b\|^2$. For the solution $x^* = A^+ b$, where A^+ is the Moore-Penrose pseudoinverse of A, we calculate:

$$\|Ax^* - b\|^2 = \|A(A^+ b) - b\|^2 = \|AA^+ b - b\|^2 = \|Pb - b\|^2 = \|0\|^2 = 0,$$

where $P = AA^+$ is the orthogonal projection matrix onto the range of A. The operation $AA^+ b$ simplifies to Pb because AA^+ acts as the projection onto the range of A, simplifying the expression further to Pb. Since $Pb = b$ for all b in the column space of A, it implies that the residuals $Pb - b = 0$, affirming that x^* indeed minimizes the squared error to zero.

Additionally we can verify this through the gradient condition. For the function $f(x) = \|Ax - b\|_2^2$, the optimal solution must satisfy:

$$\nabla f(x) = 0.$$

The gradient of $f(x)$ with respect to x is given by:

$$\nabla f(x) = 2A^T(Ax - b).$$

Setting this to zero for the optimal condition, we have:

$$2A^T(Ax - b) = 0 \Rightarrow A^T(Ax - b) = 0.$$

Substituting $x = x^*$:

$$A^T(Ax^* - b) = A^T(AA^+b - b) = A^T(Pb - b) = A^T(0) = 0,$$

which confirms that the gradient at x^* is zero, therefore, satisfying the necessary condition for a minimum.

Thus, $x^* = A^+b$ is not only a solution that minimizes $\|Ax - b\|^2$ but also satisfies the gradient condition for optimality in an unconstrained optimization framework. This dual verification through both projection and gradient analysis robustly establishes x^* as the optimal solution.

2.3 c

Let $A \in R^{m\times n}$ and $b \in R^m$. Consider the least squares problem:

$$\min_x \|Ax - b\|^2.$$

Given a matrix $A \in R^{m\times n}$ and a vector $b \in R^m$, we aim to solve the least squares problem by finding $x^* = A^+b$, where A^+ denotes the pseudoinverse of A.

The solution $x^* = A^+b$ is optimal for the problem:

$$\min_x \|Ax - b\|^2.$$

To verify the optimality and properties of x^*, we consider two key conditions:

1. $d \in \text{Range}(C)$
2. $C^T d = A^T b$

where $d = A^T b$ and $C = A^T A$.

Using the singular value decomposition of A, express A as $A = U\Sigma V^T$, where U and V are orthogonal matrices, and Σ is a diagonal matrix of singular values.

Thus, C can be expressed as:

$$C = A^T A = V\Sigma^T \Sigma V^T.$$

And the vector d is:

$$d = A^T b = V\Sigma^T U^T b.$$

Since V spans the range of C, and d is expressed in terms of V and Σ, it follows that d is in the range of C, confirming that:

$$d \in \text{Range}(C).$$

Given that the columns of V span the range of C, d is clearly in the range of C, since:

$$\text{Range}(C) = \text{Range}(V).$$

Thus the solution $x^* = A^+ b$ is a solution to the one with the minimum 2-norm among all solutions, proving it to be the optimal solution in the sense of the 2-norm minimization.

2.4 Extra Credit

Prove that if $d \in \text{Range}(C)$ (i.e., $Cx = d$ is feasible), then $x^* = C^+ d$ is an optimal solution to the 2-norm minimization problem $\min \|x\|_2$ subject to $Cx = d$:

We prove that if d is in the range of matrix C, and the equation $Cx = d$ is feasible, then the solution $x^* = C^+ d$ provided by the pseudoinverse C^+ is the optimal solution to the 2-norm minimization problem:

$$\min \|x\|_2 \quad \text{subject to} \quad Cx = d.$$

The pseudoinverse C^+ of a matrix C has several crucial properties, notably that $C^+ C$ and CC^+ are projection matrices. These properties facilitate the optimal solution as follows:

- $C^+ C$ projects onto the range (column space) of C^T.
- CC^+ projects onto the range of C.

The operation $x^* = C^+ d$ projects d onto the column space of C^T. This is significant because it ensures that x^* lies in the intersection of the column space of C^T and the affine space defined by the equation $Cx = d$, and it is the vector in this intersection with the minimal 2-norm.

The projection can be expressed mathematically as:

$$x^* = C^+ d = C^+ C x^*.$$

This expression confirms that x^* is not only in the column space of C^T but is also the image under the projection matrix $C^+ C$, which projects vectors onto this space. This solution is the optimal 2-norm solution because it minimizes the Euclidean distance to the origin while satisfying $Cx = d$.

The pseudoinverse solution $x^* = C^+ d$ is shown to be optimal in the sense that it minimizes the 2-norm among all solutions that satisfy $Cx = d$. The proof hinges on the projection properties of C^+ and the orthogonality principle, which ensure that x^* has no components in the direction of any vectors that lie in the null space of C.

3 Problem 3C

3.1 Mathematical Expression

Given a dataset $\{(x_i, y_i)\} \subseteq R^2$ where $i = 1, 2, \ldots, m$, we aim to fit a polynomial model of degree k. The polynomial model is defined by the function:

$$y = \alpha^T \phi(x)$$

where $\Phi = [\phi_0, \phi_1, \ldots, \phi_k]^T$ is the vector of coefficients, and $x^{(k)} = [1, x, x^2, \ldots, x^k]^T$ represents the powers of x.

To express this in a matrix form suitable for least squares regression, we use our Φ above however this time in matrix form:

$$\Phi = \begin{bmatrix} x_1^k & x_1^{k-1} & \ldots & x_1 & 1 \\ x_2^k & x_2^{k-1} & \ldots & x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_m^k & x_m^{k-1} & \ldots & x_m & 1 \end{bmatrix}$$

This matrix representation allows us to model the polynomial relationship and apply least squares regression to estimate the polynomial coefficients by minimizing the squared residuals between the predicted values and the observed data.

3.2 Mathematical Expression

Generally, our regularization term is as follows:

$$\min_\alpha \|\Phi\alpha - y\|_2^2 + \mathcal{L}(\alpha)$$

And if we want to apply L2 norm coefficients in the polynomial model, we get:

$$\min \|\Phi\alpha - y\|_2^2 + \lambda\|\alpha\|_2^2$$

And the closed form solution:

$$\Phi^* = (X^T X + \lambda I)^{-1} X^T y$$

3.3 Code

```python
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import cvxpy as cp

# Generate the data set, containing 100 instances
np.random.seed(seed=0)
data_size = 100
```

```
data_x = np.linspace(0, 2*np.pi, num=data_size)
data_y = np.apply_along_axis(lambda x: np.sin(x), 0, data_x) + 0.25 * np.random.normal(size=data_x.shape)

# A figure of the underlying generator function and the data points
plt.scatter(data_x, data_y, color="black")
xlist = np.linspace(0, 2*np.pi, num=100)
plt.plot(xlist, np.apply_along_axis(lambda x: np.sin(x), 0, xlist), color="green")

# Splitting the training and test set
X_train, X_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2, random_state=0)

# Define the degree of the polynomial
k = 3

# Create a Vandermonde matrix for X_train
n = len(X_train)
X_vander = np.vander(X_train, N=k+1, increasing=True)

# Define the CVXPY variables for polynomial coefficients
coeffs = cp.Variable(k+1)

# Define the objective function (least squares)
objective = cp.Minimize(cp.sum_squares(X_vander @ coeffs - y_train))

# Define the problem and solve
problem = cp.Problem(objective)
problem.solve(solver=cp.SCS)

# Output the learned polynomial coefficients
print("The polynomial coefficients are:", coeffs.value)

# Generate a dense grid of x values for plotting the fitted polynomial
x_dense = np.linspace(0, 2*np.pi, 400)
X_vander_dense = np.vander(x_dense, N=k+1, increasing=True)

# Evaluate the polynomial at the grid points
y_dense = X_vander_dense @ coeffs.value

plt.scatter(X_train, y_train, color="black", label="Data points")
plt.plot(x_dense, y_dense, label=f"Fitted Polynomial (degree {k})", color="blue")
plt.legend()
plt.show()
```

3.4 Code

```python
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import cvxpy as cp

# Generate the data set, containing 100 instances
np.random.seed(seed=0)
data_size = 100
data_x = np.linspace(0, 2*np.pi, num=data_size)
data_y = np.apply_along_axis(lambda x: np.sin(x), 0, data_x) + 0.25 * np.random.normal(size=data_x.shape)

# A figure of the underlying generator function and the data points
plt.scatter(data_x, data_y, color="black")
xlist = np.linspace(0, 2*np.pi, num=100)
plt.plot(xlist, np.apply_along_axis(lambda x: np.sin(x), 0, xlist), color="green")

# Splitting the training and test set
X_train, X_test, y_train, y_test = train_test_split(data_x, data_y, test_size=0.2, random_state=0)

# Define the degree of the polynomial
k = 12  # Degree of the polynomial

# Create a Vandermonde matrix for X_train
X_vander = np.vander(X_train, N=k+1, increasing=True)

# Define the CVXPY variables for polynomial coefficients
coeffs = cp.Variable(k+1)

# Define the regularization parameter
lambda_reg = 1

# Define the objective function (least squares + L1 regularization)
objective = cp.Minimize(cp.sum_squares(X_vander @ coeffs - y_train) + lambda_reg * cp.norm1(coeffs))

# Define the problem and solve
problem = cp.Problem(objective)
problem.solve(solver=cp.SCS)

# Output the learned polynomial coefficients
print("The polynomial coefficients are:", coeffs.value)

# Generate a dense grid of x values for plotting the fitted polynomial
x_dense = np.linspace(0, 2*np.pi, 400)
X_vander_dense = np.vander(x_dense, N=k+1, increasing=True)

```

```python
# Evaluate the polynomial at the grid points
y_dense = X_vander_dense @ coeffs.value

plt.scatter(X_train, y_train, color="black", label="Data points")
plt.plot(x_dense, y_dense, label=f"Fitted Polynomial (degree {k})", color="blue")
plt.legend()
plt.show()

# Calculate and print the test error
X_vander_test = np.vander(X_test, N=k+1, increasing=True)
test_error = np.mean((X_vander_test @ coeffs.value - y_test) ** 2)
print("Test error:", test_error)

# Check the sparsity of the optimal solution
sparsity = np.sum(coeffs.value == 0)
print("Number of zero coefficients (sparsity):", sparsity)
```

3.5 Extra credit

NA

4 Problem 3D

4.1 Mathematical Expression Code

The mathematical expression for the minimization problem in the Support Vector Machine (SVM) model is given by:

$$\min_{w,\beta} \frac{1}{2}\|w\|^2$$

subject to the constraint that for all training samples,

$$y_i(w^T x_i + \beta) \geq 1, \quad \forall i = 1, 2 \ldots m$$

Here, w is the weight vector and β (often denoted as b and misread as η) is the bias term of the hyperplane. The vectors x_i represent the feature vectors of the training samples, and y_i are the labels of these samples, which are typically $+1$ or -1, corresponding to the two classes.

```python
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
solver=cp.ECOS
```

```python
WDBC = datasets.load_breast_cancer()
X, y = datasets.load_breast_cancer(return_X_y=True)
# Changing labels
y = 2*y - 1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Normalize the features on the training set to have mean 0 and standard deviation 1
for i in range(X_train.shape[1]):
    X_train_mean = np.mean(X_train[:,i])
    X_train_std = np.std(X_train[:,i])
    X_train[:,i] = (X_train[:,i] - X_train_mean)/X_train_std
    X_test[:,i] = (X_test[:,i] - X_train_mean)/X_train_std


# 2-dimensional example
X2D_train = X_train[:,:4]
#X2D_test = X_test[:,:40]
X2D_test = X_test[:, :4]

fig, ax = plt.subplots()
ax.scatter(X2D_train[:,0], X2D_train[:,1], c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors="k")
plt.show()


import cvxpy as cp

# Define the variables
d = X2D_train.shape[1]
w = cp.Variable(d)
beta = cp.Variable()

# Set up the problem
objective = cp.Minimize(0.5 * cp.norm(w, 2)**2)
constraints = [y_train * (X2D_train @ w + beta) >= 1]
prob = cp.Problem(objective, constraints)

# Solve the problem
prob.solve()

# Print results
w_value = w.value
beta_value = beta.value
print("Weight vector (w):", w_value)
print("Bias (beta):", beta_value)

# Predicting test labels
predictions = X2D_test @ w_value + beta_value
```

```
predictions = np.where(predictions >= 0, 1, -1)

# Calculate test error
test_error = np.mean(predictions != y_test)
print("Test error:", test_error)
Weight vector (w): [4.21618501e-10 2.34296821e-10 4.28819818e-10 4.06486254e-10]
Bias (beta): 0.10956616072203625
Test error: 0.3706293706293706]
```

4.2 Mathematical Expression and Code

Mathematical Expression Let:

- x_i be the input features for each data point i,
- y_i be the corresponding labels, which are ± 1,
- w be the weight vector of the hyperplane,
- β (often denoted as b) be the bias term,
- ξ_i be the slack variables representing the degree of misclassification for each data point i.

$$\min_{w,\beta,\xi} \left(\frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i \right) \tag{1}$$

s.t

$$y_i(w^T x_i + \beta) \geq 1 - \xi_i \quad \text{for all } i \tag{2}$$

and

$$\xi_i \geq 0 \quad \text{for all } i \tag{3}$$

where C is a regularization parameter that controls the trade-off between achieving a low error on the training data and maintaining a small norm for w.

```
import numpy as np
import cvxpy as cp
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

#Data Prepration
WDBC = datasets.load_breast_cancer()
X, y = datasets.load_breast_cancer(return_X_y=True)
# Change labels from {0, 1} to {-1, 1}
y = 2 * y - 1
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

for i in range(X_train.shape[1]):
    X_train_mean = np.mean(X_train[:, i])
    X_train_std = np.std(X_train[:, i])
    X_train[:, i] = (X_train[:, i] - X_train_mean) / X_train_std
    X_test[:, i] = (X_test[:, i] - X_train_mean) / X_train_std

fig, ax = plt.subplots()
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors="k")
plt.show()

# Setup SVM model
d = X_train.shape[1]
w = cp.Variable(d)
beta = cp.Variable()
e = cp.Variable(X_train.shape[0])
lambda_param = 1

# Objective and constraints
objective = cp.Minimize(cp.norm(e, 1) + lambda_param * cp.norm(w, 2)**2)
constraints = [y_train * (X_train @ w + beta) + e >= 1, e >= 0]
prob = cp.Problem(objective, constraints)

prob.solve(solver=cp.ECOS)

print("Weight vector (w):", w.value)
print("Bias (beta):", beta.value)

predictions = X_test @ w.value + beta.value
predictions = np.where(predictions >= 0, 1, -1)

test_error = np.mean(predictions != y_test)
print("Test error:", test_error)
Weight vector (w): [ 3.38538449e-10  1.88052705e-10  3.44325806e-10  3.26374870e-10
  1.75975354e-10  2.86678687e-10  3.22528516e-10  3.64615422e-10
  1.61129917e-10 -1.09939421e-11  2.61106206e-10 -1.54766486e-11
  2.55501786e-10  2.46934770e-10 -4.21915318e-11  1.24379553e-10
  8.76699178e-11  1.75935092e-10 -1.39383048e-11  2.51790383e-11
  3.61229503e-10  2.08667935e-10  3.63913517e-10  3.39106418e-10
  2.02980273e-10  2.83722806e-10  3.09453603e-10  3.73915076e-10
  1.99963993e-10  1.51908233e-10]
Bias (beta): 0.07291137243980425
Test error: 0.3706293706293706
```