Assessed Coursework 4

CID 01252821

Question 1

Job 1

```
from os.path import join

import numpy as np
from pyspark import SparkContext

sc = SparkContext(appName="a4jtl22q1")

HDFS_PATH = "hdfs://howitzer:9000/shared_data"

text_file = sc.textFile(join(HDFS_PATH, "breast_cancer_train.csv"))
# filter out the header
lines = text_file.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0])

def compute_outer_product(line):
    line = line.strip().split(",")
    data_line = np.array([float(x) for x in line[1:]])
    outer_prod = np.outer(data_line, data_line).flatten()
    out1 = '_'.join([str(x) for x in data_line])
    out2 = '_'.join([str(x) for x in outer_prod])
    return (line[0], out1, out2, 1)

computed_data = lines.map(compute_outer_product)

def parse_line(line):
    key, sum_x, outer_x, weight_x = line
    weight_x = float(weight_x)
    sum_x = np.array([float(x.strip()) for x in sum_x.split('_')])
    outer_x = np.array([float(x.strip()) for x in outer_x.split('_')])

    return (key, (sum_x, outer_x, weight_x))

parsed_data = computed_data.map(parse_line)

def reducer_fn(x, y):
    sum_x1, outer_x1, weight_x1 = x
    sum_x2, outer_x2, weight_x2 = y

    return (sum_x1 + sum_x2, outer_x1 + outer_x2, weight_x1 + weight_x2)
```

```
reduced_data = parsed_data.reduceByKey(reducer_fn)

def compute_final_output(x):
    key, (sum_x, outer_x, weight_x) = x
    mean_x = sum_x / weight_x
    mean_out = '_'.join([str(e) for e in mean_x])
    cov_x = outer_x / weight_x - np.outer(mean_x, mean_x).flatten()
    cov_out = '_'.join([str(e) for e in cov_x])

    return (key, weight_x, mean_out, cov_out)

final_output = reduced_data.map(compute_final_output)

final_output.collect()
```

Estimated mean parameters:

$$\mu_M = (17.46, 21.63, 0.10, 0.14)$$
$$\mu_B = (12.22, 17.93, 0.09, 0.08)$$

Estimated covariances Σ_M and Σ_B:

$$\Sigma_M = \begin{bmatrix} 9.836 & 0.455 & -0.005 & 0.018 \\ 0.455 & 14.91 & -0.007 & -0.002 \\ -0.005 & -0.007 & 0.0002 & 0.0005 \\ 0.018 & -0.002 & 0.0005 & 0.003 \end{bmatrix}$$

$$\Sigma_B = \begin{bmatrix} 2.774 & -0.426 & -0.004 & 0.004 \\ -0.426 & 16.54 & -0.014 & -0.011 \\ -0.004 & -0.014 & 0.0002 & 0.0003 \\ 0.004 & -0.011 & 0.0003 & 0.001 \end{bmatrix}$$

Note: The code is taken from the published Solutions of Coursework 2 and is directly translated to RDDs.

Job 2

```
from pyspark import SparkContext
from math import log
import numpy as np
from scipy.spatial.distance import mahalanobis as maha
from scipy.linalg import inv, det

# Loading the output from previous step
f = final_output.collect()
p = {}
mu = {}
Sigma_inv = {}
log_det_Sigma = {}
for line in f:
  key = line[0]
  p[key] = float(line[1])
```

```
  mu[key] = np.array([float(x.strip()) for x in line[2].split('_')])
  Sigma_inv[key] = inv(
    np.array([float(x.strip()) for x in line[3].split('_')]).reshape((4,4)))
  log_det_Sigma[key] = -log(det(Sigma_inv[key]))

kappa = 2 * (log(p['B']) - log(p['M']))

text_file_test = sc.textFile(join(HDFS_PATH, "breast_cancer_test.csv"))
# filter out the header
data = text_file_test.zipWithIndex().filter(lambda x: x[1] > 0).map(lambda x: x[0])

# define the function that will be used in map transformation
def classify(line):
    line = line.strip().split(',')
    if line[0] != 'diagnosis': ## Ignore header
        xi = np.array([float(x) for x in line[1:]])
        pred_val = (
          maha(xi, mu['B'], Sigma_inv['B'])**2) - (maha(xi, mu['M'], Sigma_inv['M'])**2)
        if pred_val - log_det_Sigma['M'] + log_det_Sigma['B'] > kappa:
            pred_label = 'M'
        else:
            pred_label = 'B'
        return ('dummy_key', line[0] + ',' + pred_label)

# Apply the map transformation
classified_data = data.map(classify)

def parse_line(line):
    key, labels = line
    actual, predicted = labels.split(',')
    return ((actual, predicted), 1)

parsed_data = classified_data.map(parse_line)

label_counts = parsed_data.reduceByKey(lambda a, b: a + b).collectAsMap()

TN = label_counts.get(('B', 'B'), 0)
FP = label_counts.get(('B', 'M'), 0)
FN = label_counts.get(('M', 'B'), 0)
TP = label_counts.get(('M', 'M'), 0)

print('Sensitivity:', TP / (TP + FN))
print('Specificity:', TN / (TN + FP))
print('Precision:', TP / (TP + FP))
print('Accuracy:', (TP + TN) / (TP + TN + FP + FN))
sc.stop()
```

```
Sensitivity: 0.9047619047619048
Specificity: 0.9385964912280702
Precision: 0.8444444444444444
Accuracy: 0.9294871794871795
```

Note: The code is taken from the published Solutions of Coursework 2 and is directly translated to RDDs.

Question 2

```
import numpy as np
from math import pi
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.linalg import Vectors
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, cos, sin, lit

spark = SparkSession.builder.appName("a4-q2-jtl22").getOrCreate()
```

Part a

```
# change it to read directly from HDFS
df = (
  spark.read.option("header", True)
  .option("inferSchema", True)
  .option("delimiter", ";")
  .csv(join(HDFS_PATH, "mauna_loa_co2.csv"))
)

# create necessary columns to DataFrame
df = df.withColumn('ones', lit(1.0))
df = df.withColumn('x', (df['date'] - 1958))
df = df.withColumn('x_squared', df['x'] ** 2)
df = df.withColumn('cos_x', cos(2 * pi * df['x']))
df = df.withColumn('sin_x', sin(2 * pi * df['x']))

# combine the transformed variables into a DenseVector
vecAssembler = VectorAssembler(
  inputCols=['x', 'x_squared', 'cos_x', 'sin_x'], outputCol='features')

# Transform the DataFrame
df = vecAssembler.transform(df)

# Keep only necessary columns and show the DataFrame
df_fit = df.select(['date', 'batch', 'CO2', 'features'])
```

Here, I am not including the intercept column, instead a fit with intercept set to True will be used. This is equivalent to adding a column with values 1 and fitting a model with intercept set to False.

Part b

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# split data into training and test set
train = df_fit.filter(df['date'] < 2015)
test = df_fit.filter(df['date'] >= 2015)
```

```
# Linear Regression estimator
lr = LinearRegression(
    featuresCol='features', labelCol='CO2', fitIntercept=True)

# Fit the model to the training data
lr_model = lr.fit(train)

# Get the coefficients
coefficients = lr_model.coefficients
intercept = lr_model.intercept

# Print the coefficients
print(f"Intercept: {intercept}")
print(f"Coefficients: {coefficients}")

# Make predictions on the test data
predictions = lr_model.transform(test)

# Initialize a regression evaluator
evaluator = RegressionEvaluator(
  predictionCol="prediction", labelCol="CO2", metricName="rmse")

# Calculate and print the root mean squared error
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE) on test data = {rmse}")
```

```
Intercept: 314.025725720318
Coefficients: [0.810528234341543,0.012116193819406719,-1.0148643626284,2.6209466204801846]
Root Mean Squared Error (RMSE) on test data = 1.868167448763008
```

Part c

```
# Get the summary of the Linear Regression model
summ = lr_model.summary

# Get the standard errors, t-values and p-values
std_errors = summ.coefficientStandardErrors
t_values = summ.tValues
p_values = summ.pValues

# Get the adjusted R2 score
adj_r2 = summ.r2adj

# Print the standard errors, t-values, p-values and adjusted R2 score
print(f"Standard Errors: {std_errors}")
print(f"T-Values: {t_values}")
print(f"P-Values: {p_values}")
print(f"Adjusted R2: {adj_r2}")

max_t_value_index = t_values.index(max(t_values))
# parameter with the lowest p-value
```

```
if max_t_value_index == 4:
    print("Intercept has the lowest p-value.")
else:
    print(f"Parameter theta_{max_t_value_index + 1} has the lowest p-value.")
```

The last elements of the output values correspond to the intercept in the following output:

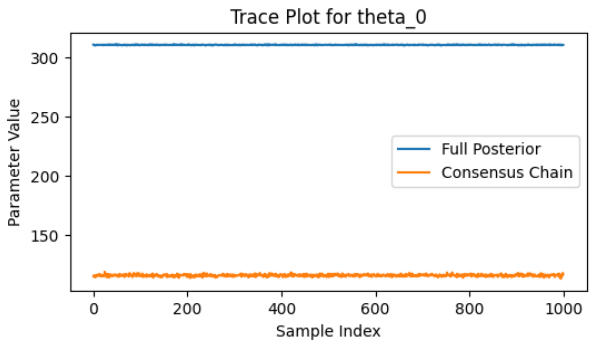
```
Standard Errors: [0.008675501562338902, 0.0001469984014340705,
0.05011306197162435, 0.049999753644764514, 0.1074140030417945]
T-Values: [
93.42724781009962, 82.4239835345481, -20.251493776274323, 52.41919068444499, 2923.508265473836]
P-Values: [0.0, 0.0, 0.0, 0.0, 0.0]
Adjusted R2: 0.9986245447463505
Intercept has the lowest p-value.
```

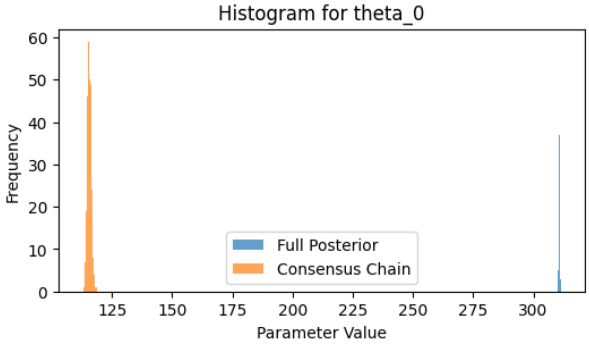
Part d

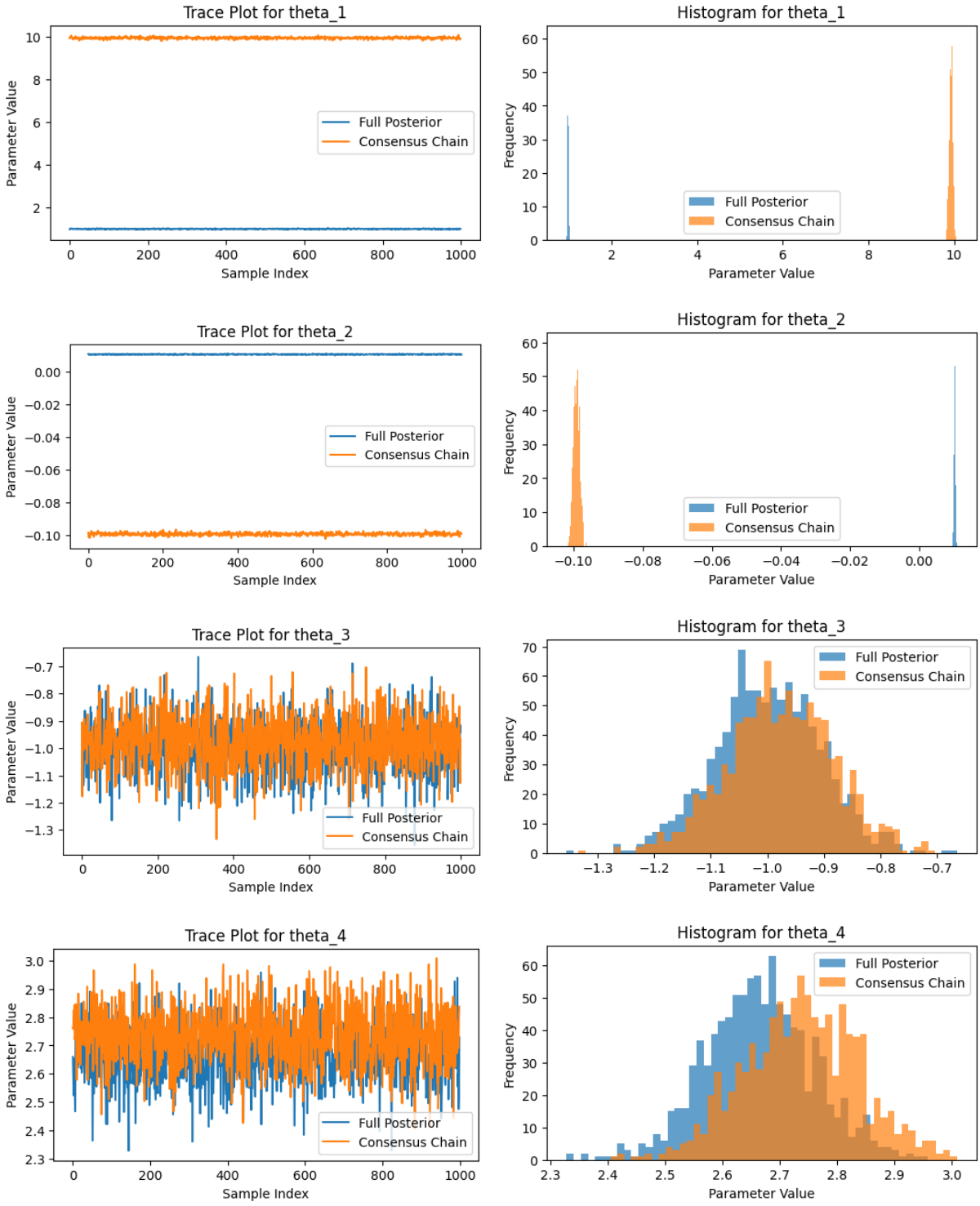
```
import pandas as pd
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt

# define constants
sigma_sq = 3.5
p = 5  #number of parameters
num_samples = 1000
sigma_0 = sigma_sq * np.eye(p)
sigma_0_inv = np.linalg.inv(sigma_0)

# PySpark DataFrame to pandas DataFrame
df_pandas = df_fit.toPandas()

# partition by batches
batches = [df_pandas[df_pandas['batch'] == i] for i in range(3)]

# calculate subposteriors for each batch
subposteriors = []
for batch in batches:
    # design matrix X and the response vector y
    X = np.vstack(batch['features'].values)
    X = np.c_[np.ones(len(batch)), X] # append the 1 column
    y = batch['CO2'].values

    # posterior parameters
    Sigma = np.linalg.inv(sigma_0_inv + (1 / sigma_sq) * X.T @ X)
    mu = (1 / sigma_sq) * Sigma @ X.T @ y

    # sample from the subposterior
    subposterior = multivariate_normal.rvs(mean=mu, cov=Sigma, size=num_samples)
    subposteriors.append(subposterior)

# consensus posterior
```

```
consensus_posterior = np.mean(subposteriors, axis=0)

# design matrix X and the response vector y for full-data posterior calculation
X = np.vstack(df_pandas['features'].values)
X = np.c_[np.ones(len(df_pandas)), X]
y = df_pandas['CO2'].values

# full-data posterior parameters
Sigma = np.linalg.inv(sigma_0_inv + (1 / sigma_sq) * X.T @ X)
mu = (1 / sigma_sq) * Sigma @ X.T @ y

# sample from the full-data posterior
full_data_posterior = multivariate_normal.rvs(mean=mu, cov=Sigma, size=num_samples)

for i in range(p):
    full = full_data_posterior[:, i]
    consensus = consensus_posterior[:, i]

    # trace plot
    plt.figure(figsize=(6, 3))
    plt.plot(full, label='Full Posterior')
    plt.plot(consensus, label='Consensus Chain')
    plt.xlabel('Sample Index')
    plt.ylabel('Parameter Value')
    plt.title(f'Trace Plot for theta_{i}')
    plt.legend()
    plt.show()

    # histograms
    plt.figure(figsize=(6, 3))
    plt.hist(full, bins=50, alpha=0.7, label='Full Posterior')
    plt.hist(consensus, bins=50, alpha=0.7, label='Consensus Chain')
    plt.xlabel('Parameter Value')
    plt.ylabel('Frequency')
    plt.title(f'Histogram for theta_{i}')
    plt.legend()
    plt.show()
```

```
spark.stop()
```







Question 3

Part a i)

By problem statement $y_i|x_i, \theta \sim N(x_i^T\theta, 1)$, which is equvialent to

$$y_i = x_i^T\theta + \epsilon_i, \text{ where } \epsilon_i \sim N(0,1).$$

Our aim is to show that the estimator

$$\hat{\theta}_n(\lambda) = [S_n^{(xx)}(\lambda)]^{-1} S_n^{(xy)}(\lambda)$$

is unbiased, or in other words to prove that $E(\hat{\theta}_n(\lambda)) = \theta$. Let's recursively expand the following:

$$S_n^{(xx)}(\lambda) = \lambda S_{n-1}^{(xx)}(\lambda) + x_n x_n^T =$$
$$= \lambda^2 S_{n-2}^{(xx)}(\lambda) + \lambda x_n x_n^T + x_n x_n^T =$$
$$...$$
$$= \lambda^n S_0^{(xx)}(\lambda) + \lambda^{n-1} x_n x_n^T + ... + \lambda x_n x_n^T + x_n x_n^T =$$
$$= \lambda^{n-1} x_n x_n^T + ... + \lambda x_n x_n^T + x_n x_n^T,$$

and

$$S_n^{(xy)}(\lambda) = \lambda^{n-1} x_n y_n + ... + \lambda x_n y_n + x_n y_n$$

Now, let's use $y_i = x_i^T\theta + \epsilon_i$:

$$S_n^{(xy)}(\lambda) = \lambda^{n-1} x_n y_n + ... + \lambda x_n y_n + x_n y_n =$$

$$= \lambda^{n-1} x_n (x_n^T\theta + \epsilon_n) + ... + \lambda x_n (x_n^T\theta + \epsilon_n) + x_n (x_n^T\theta + \epsilon_n) =$$

$$= \theta S_n^{(xx)}(\lambda) + x_n \epsilon_n (\lambda^{n-1} + ... + \lambda + 1).$$

For the estimator we get:

$$\hat{\theta}_n(\lambda) = [S_n^{(xx)}(\lambda)]^{-1} S_n^{(xy)}(\lambda) =$$

$$= [S_n^{(xx)}(\lambda)]^{-1} \left[\theta S_n^{(xx)}(\lambda) + x_n \epsilon_n (\lambda^{n-1} + ... + \lambda + 1)\right] =$$
$$= \theta + [S_n^{(xx)}(\lambda)]^{-1} x_n \epsilon_n (\lambda^{n-1} + ... + \lambda + 1)$$

or

$$E\left[\hat{\theta}_n(\lambda)\right] = E\left[\theta + [S_n^{(xx)}(\lambda)]^{-1} x_n \epsilon_n (\lambda^{n-1} + ... + \lambda + 1)\right] =$$

$$= E[\theta] + E[[S_n^{(xx)}(\lambda)]^{-1} x_n \epsilon_n \lambda^{n-1}] + ... + E[[S_n^{(xx)}(\lambda)]^{-1} x_n \epsilon_n \lambda] + E[[S_n^{(xx)}(\lambda)]^{-1} x_n \epsilon_n] =$$
$$= E[\theta] + x_n \lambda^{n-1} E[S_n^{(xx)}(\lambda)]^{-1}] E[\epsilon_n] + ... + x_n \lambda E[S_n^{(xx)}(\lambda)]^{-1}] E[\epsilon_n] + x_n E[S_n^{(xx)}(\lambda)]^{-1}] E[\epsilon_n] = \theta,$$

hence, the estimator $\hat{\theta}_n(\lambda)$ is unbiased.

Part a ii)

By problem statement $S_n^{(xx)}(\lambda) \in R^{p\times p}$ and $\lambda \in (0,1)$, therefore $\lambda S_n^{(xx)}(\lambda) \in R^{p\times p}$.

$$\hat{\theta}_n(\lambda) = [S_n^{(xx)}(\lambda)]^{-1} S_n^{(xy)}(\lambda) = [\lambda S_{n-1}^{(xx)}(\lambda) + x_n x_n^T]^{-1} S_n^{(xy)}(\lambda)$$

We can apply the Sherman-Morrison formula to $\lambda S_{n-1}^{(xx)}(\lambda) + x_n x_n^T$:

$$[\lambda S_{n-1}^{(xx)}(\lambda) + x_n x_n^T]^{-1} = [\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} - \frac{[\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} x_n x_n^T [\lambda S_{n-1}^{(xx)}(\lambda)]^{-1}}{1 + x_n^T [\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} x_n},$$

therefore

$$\hat{\theta}_n(\lambda) = [S_n^{(xx)}(\lambda)]^{-1} S_n^{(xy)}(\lambda) = [\lambda S_{n-1}^{(xx)}(\lambda) + x_n x_n^T]^{-1} S_n^{(xy)}(\lambda) =$$

$$= \left[[\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} - \frac{[\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} x_n x_n^T [\lambda S_{n-1}^{(xx)}(\lambda)]^{-1}}{1 + x_n^T [\lambda S_{n-1}^{(xx)}(\lambda)]^{-1} x_n} \right] S_n^{(xy)}(\lambda).$$

This formula will use the already calculated inverted matrix from the previous step, and in the current we can save the value for S_{n-1} which we just calculated to be used in the next step.

Part b

$y = X\theta + \epsilon$, where $\epsilon \sim N(N_n(0, \sigma^2 I_n))$ implies

$$l(\theta) = (X\theta - y)^T (X\theta - y),$$

and

$$\nabla_\theta l(\theta) = 2(X^T X\theta - X^T y).$$

Therefore the the mini-batch stochastic gradient descent update for θ is

$$\theta^{(k+1)} = \theta^{(k)} - \rho_k \hat{\Lambda}'_j(\tilde{\theta}) =$$

$$= \theta^{(k)} - \frac{2\rho_k}{m} \sum_{i=1}^{m} (X_{j_i^{(k)}}^T X_{j_i^{(k)}} \theta^k - X^T y_{j_i^{(k)}]}),$$

where $j_i^{(k)}$ are $m.$ indices sampled from $1, ...n$ at the k-th step. The algorithm is synchronous.

Part c

Question 4

Data Source: https://archive.ics.uci.edu/dataset/265/physicochemical+properties+of+protein+tertiary+structure

This is a data set of Physicochemical Properties of Protein Tertiary Structure. The data set is taken from CASP 5-9. There are 45730 decoys and size varying from 0 to 21 armstrong.

Attributes Information:

- RMSD - Size of the residue.
- F1 - Total surface area.
- F2 - Non polar exposed area.
- F3 - Fractional area of exposed non polar residue.
- F4 - Fractional area of exposed non polar part of residue.
- F5 - Molecular mass weighted exposed area.
- F6 - Average deviation from standard exposed area of residue.
- F7 - Euclidean distance.
- F8 - Secondary structure penalty.
- F9 - Spacial Distribution constraints (N,K Value).

Root-mean-square-deviation (RMSD) is an indicator in protein-structure-prediction-algorithms (PSPAs). The question we are trying to answer is whether we can model the RMSD well using only linear relationships?

For the purpose of just illustrating a workflow using PySpark and having reasonable visualizations, I am going to work with a randomly seeded sample of 1% of the data. The data is also split in train and test sets with a 9:1 ratio. The reported metrics and the predictions visualizations are using the test set. The distribution of the RMSD data can be seen in Figure 1, Figure 2 shows a heatmap where we can see the correlations between RMSD and each of the features which are relatively low, with F3 showing the most promise. This means that it will likely be hard to make good predictions only with linear features. In Figure 3 we can see the distirbutions of the relationships between the variables, extending the information obtained from Figure 2.

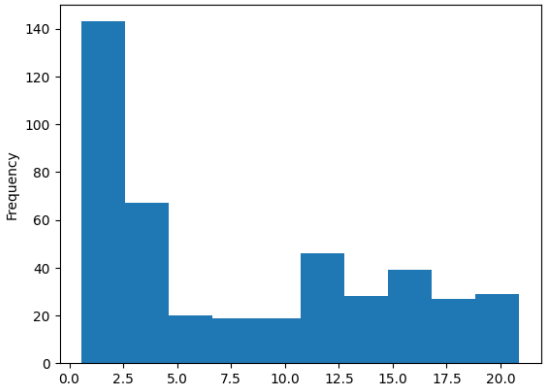


Figure 1: RMSD distribution

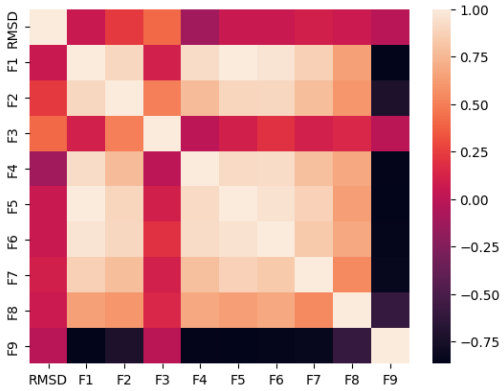


Figure 2: Heatmap

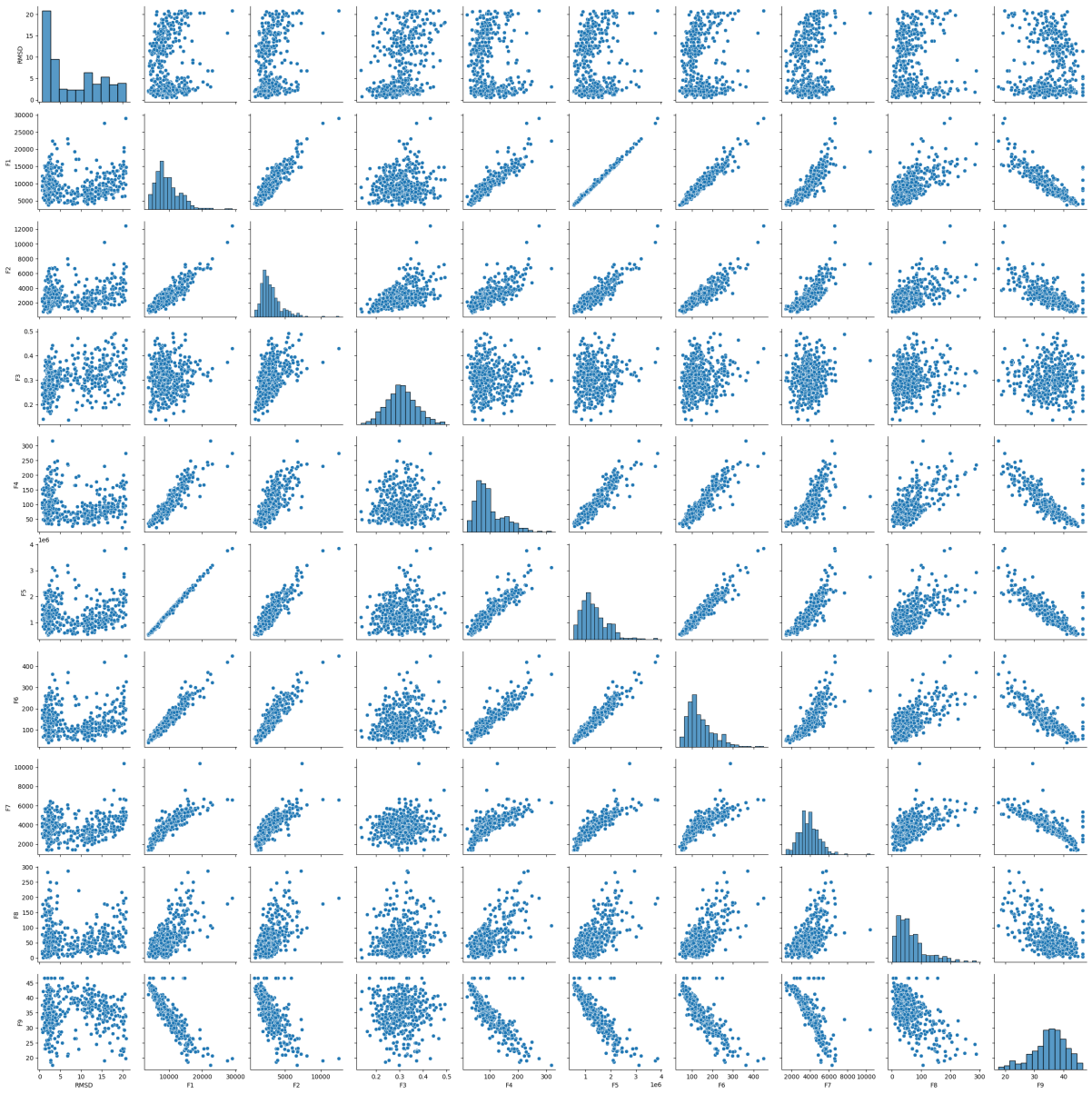


Figure 3: Variables Relationships

Three sets of models are explored, with different regularization parameters (0, 0.001, 0.01, 0.1, 1, 2, 5, 7, 10). They are fit without intercept because the side of the residue (RMSD) is logical to be 0 if all of the other features are 0. The models are compared based on RMSE.

- Linear Regression (Ordinary Least Squares)
- Lasso Regression
- Ridge Regression

Results showing the obtained RMSE based on the regularisation parameter:

Model	0	0.001	0.01	0.1	1	2	5	7	10
OLS	6.372	X	X	X	X	X	X	X	X
Lasso	6.372	6.373	6.326	6.253	6.374	6.431	6.633	6.792	7.098
Ridge	6.372	6.372	6.370	6.307	6.145	6.160	6.250	6.297	6.350

The best RMSE is for Ridge Regression with `regParam = 1` and is `rmse = 6.145`. Just to confirm that our intuition is correct that we don't need an intercept parameter, the chosen model was ran with intercept and the resulting rmse was `rmse = 6.152`, proving that the intercept is useless for our data.

If we explore this specific model we can obtain the following statistics:

```
Coefficients: [0.0001, 0.0009, 25.1499, -0.0563, 0.0, -0.0014, 0.0003, 0.0003, 0.0079, -0.0299]
T-Values: [0.47, 2.52, 5.03, -5.60, 0.41, -0.16, 0.61, 0.61, 1.21, -0.85]
P-Values: [
0.64, 0.01, 7.680661477937889e-07, 4.0992752525781384e-08, 0.68, 0.88, 0.54, 0.54, 0.23, 0.40
]
Adjusted R2: 0.7085593784306647
```

F2-F4 are significant based on the p-values. This also confirms that F3 is indeed important, combined with the fact that the estimated coefficient for F3 is 25 compared to all other coefficients which are around 0. Based on the p-values $F4$ seems tobe the most significant feature. Overall, the results are not very promising getting an RMSE of `6.145` when the mean of the RMSD is only `8.1`. Figure 4 shows the RMSD true values and the corresponding predictions, confirming that RMSD can't be modelled well with just linear relationships.

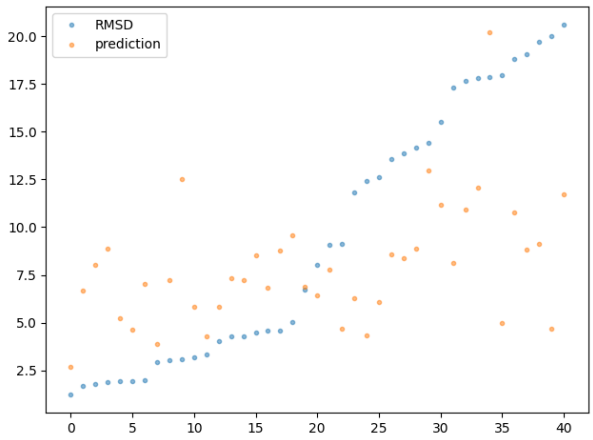


Figure 4: Predictions vs True values

Code:

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

spark = SparkSession.builder.getOrCreate()

# load data
df = spark.read.format("csv").option("header","true").option("inferSchema", "true").load("CASP.csv")
# drop NA values
df = df.na.drop()
df = df.sample(fraction=0.01, seed=18)
df_plots = df.toPandas()
df.show(10)

df_pandas = df.toPandas()["RMSD"]
df_pandas.plot(style='.-', figsize=(77,6))
plt.show()

# pairplot
sns.pairplot(df_plots)
plt.show()
# heatmap
sns.heatmap(df_plots.corr())
plt.show()


# Define features and target
features = ["F1", "F2", "F3", "F4", "F5", "F6", "F7", "F7", "F8", "F9"]
target = "RMSD"

# crate a single features vector
assembler = VectorAssembler(inputCols=features, outputCol="features")
df = assembler.transform(df)

# create train and test sets
train_df, test_df = df.randomSplit([0.9, 0.1], seed=18)

# # define evaluator
evaluator = RegressionEvaluator(
  labelCol=target, predictionCol="prediction", metricName="rmse")

# Linear Regression
lr = LinearRegression(labelCol=target, fitIntercept=False)
lr_model = lr.fit(train_df)
lr_predictions = lr_model.transform(test_df)
lr_rmse = evaluator.evaluate(lr_predictions)
# Get the coefficients
lr_predictions.show()
print(f"Intercept: {lr_model.intercept}")
```

```
print(f"Coefficients: {lr_model.coefficients}")
print(f"Linear Regression RMSE: {lr_rmse}")

reg_params_values = [0, 0.001, 0.01, 0.1, 1, 2, 5, 7, 10]
# Lasso Regression
rmses = []
for reg_value in reg_params_values:
  lasso = LinearRegression(
    labelCol=target, elasticNetParam=1.0, regParam=reg_value, fitIntercept=False)
  lasso_model = lasso.fit(train_df)
  lasso_predictions = lasso_model.transform(test_df)
  lasso_rmse = evaluator.evaluate(lasso_predictions)
  rmses.append((reg_value, lasso_rmse))
print(rmses)
# Ridge Regression
rmses = []
for reg_value in reg_params_values:
  ridge = LinearRegression(
    labelCol=target, elasticNetParam=0.0, regParam=reg_value, fitIntercept=False)
  ridge_model = ridge.fit(train_df)
  ridge_predictions = ridge_model.transform(test_df)
  ridge_rmse = evaluator.evaluate(ridge_predictions)
  rmses.append((reg_value, ridge_rmse))
print(rmses)

# Explore the chosen best performing model
ridge = LinearRegression(
  labelCol=target, elasticNetParam=0.0, regParam=1, fitIntercept=False)
ridge_model = ridge.fit(train_df)
print(f"Estimated coefficients: {ridge_model.coefficients}")
ridge_predictions = ridge_model.transform(test_df)

summ = ridge_model.summary
# Get the standard errors, t-values and p-values
std_errors = summ.coefficientStandardErrors
t_values = summ.tValues
p_values = summ.pValues
# Get the adjusted R2 score
adj_r2 = summ.r2adj
# metrics
print(f"Standard Errors: {std_errors}")
print(f"T-Values: {t_values}")
print(f"P-Values: {p_values}")
print(f"Adjusted R2: {adj_r2}")

# plot predictions vs true values
res = ridge_predictions.toPandas()
res[["RMSD", "prediction"]].plot(alpha=0.5, style=".")
plt.tight_layout()
```