Million Song Dataset Prediction

Outline

Overview

Problem Statement:

- More and more music platform plan to use machine learning methods to recommend songs to users to derive financial gain
- We wants to thorough some audio features to predict the re lease year of a song and discover their relation between them
- This is a regression problem.

Overview

Dataset:

- Derive from UC Irvine Machine Learning Repository
- Totally 515345 samples in dataset, the first column is the target (actual release year of the song) the next 12 columns are timbre average, the last 78 columns are timbre covariance
- Timbre is the distinguishing characteristic that differentiates one sound from another.


```
| 2001|49.94357|21.47114|73.0775|8.74861|-17.40628|-13.09905|-25.01202|-12.23257|7.83089|-2.46783|3.32136|-2.31521|10.20556|6
11.10913|951.0896|698.11428|408.98485|383.70912|326.51512|238.11327|251.42414|187.17351|100.42652|179.19498|-8.41558|-317.870
38|95.86266|48.10259|-95.66303|-18.06215|1.96984|34.42438|11.7267|1.3679|7.79444|-0.36994|-133.67852|-83.26165|-37.29765|73.0
4667|-37.36684|-3.13853|-24.21531|-13.23066|15.93809|-18.60478|82.15479|240.5798|-10.29407|31.58431|-25.38187|-3.90772|13.292
58|41.5506|-7.26272|-21.00863|105.50848|64.29856|26.08481|-44.5911|-8.30657|7.93706|-10.7366|-95.44766|-82.03307|-35.59194|4.
69525|70.95626|28.09139|6.02015|-37.13767|-41.1245|-8.40816|7.19877|-8.60176|-5.90857|-12.32437|14.68734|-54.32125|40.14786|1
3.0162|-54.40548|58.99367|15.37344|1.11144|-23.08793|68.40795|-1.82223|-27.46348|2.26327|
```


Goal:

- Explore and visualize the data
- Develop models to predict the release year of a song
- Provide the performance evaluation of fitted models and make conclusion.

Outline

Introduction

- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion

- Summary
- Model Comparation
- Lessons Learned

Task flow

- Step1:Explorative Analysis And Visualization
- Step2:Self-Implemented Modeling
  - Linear Regression
  - Random Forest Regressor
  - Decision Tree Regressor
- Step3:Spark MLlib Modeling
  - Decision Tree Regressor
  - Random Forest Regressor
  - Linear Regression
- Step4:Tensor Flow Keras Modeling
  - Linear Regression
  - Convolutional Neural Network
  - Self-Implemented Modeling
- Step5:Summary
  - Learned Lessons
  - Interesting Finding

Outline

Preprocessing:

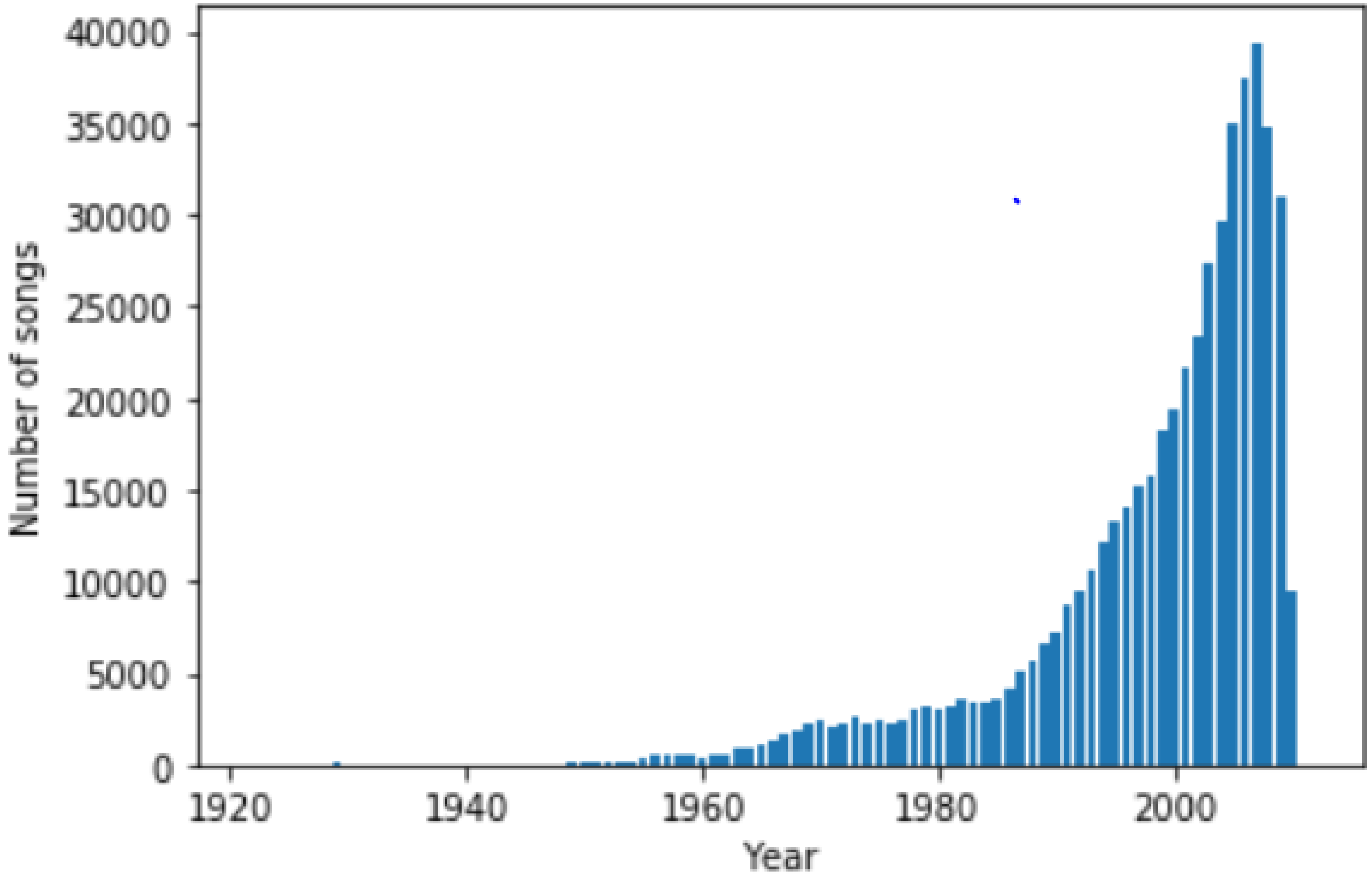
- We ignore the process of preprocessing because the dataset was already preprocessed, so it should not have noise, outliers and duplicate data

- Also we have a check it didn't have missing values

```
#check missing values
dataFrame.agg(*[(1-(fn.count(c) /fn.count('*'))).alias(c) for c in dataFrame.columns]).show()
```

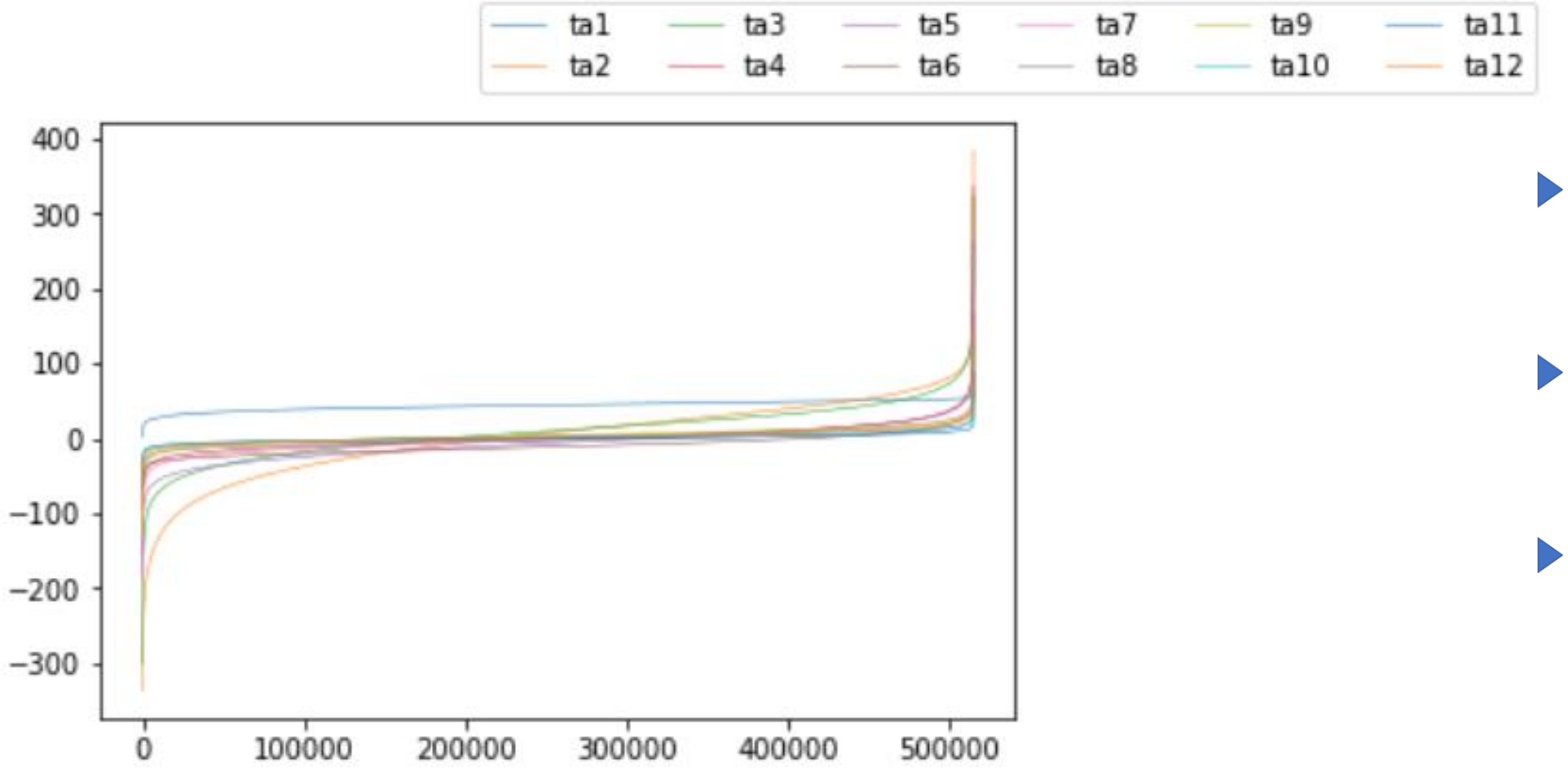
```
+-----+---+---+---+---+---+---+---+---+---+----+----+----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|label|ta1|ta2|ta3|ta4|ta5|ta6|ta7|ta8|ta9|ta10|ta11|ta12| t1| t2| t3| t4| t5| t6| t7| t8| t9|t10|t11|t12|t13|t14|t15|t16|t17|t
18|t19|t20|t21|t22|t23|t24|t25|t26|t27|t28|t29|t30|t31|t32|t33|t34|t35|t36|t37|t38|t39|t40|t41|t42|t43|t44|t45|t46|t47|t48|t49|
t50|t51|t52|t53|t54|t55|t56|t57|t58|t59|t60|t61|t62|t63|t64|t65|t66|t67|t68|t69|t70|t71|t72|t73|t74|t75|t76|t77|t78|
+-----+---+---+---+---+---+---+---+---+---+----+----+----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0| 0.0| 0.0| 0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|
0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.
0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|0.0|
```

Target Variable Balance Plot



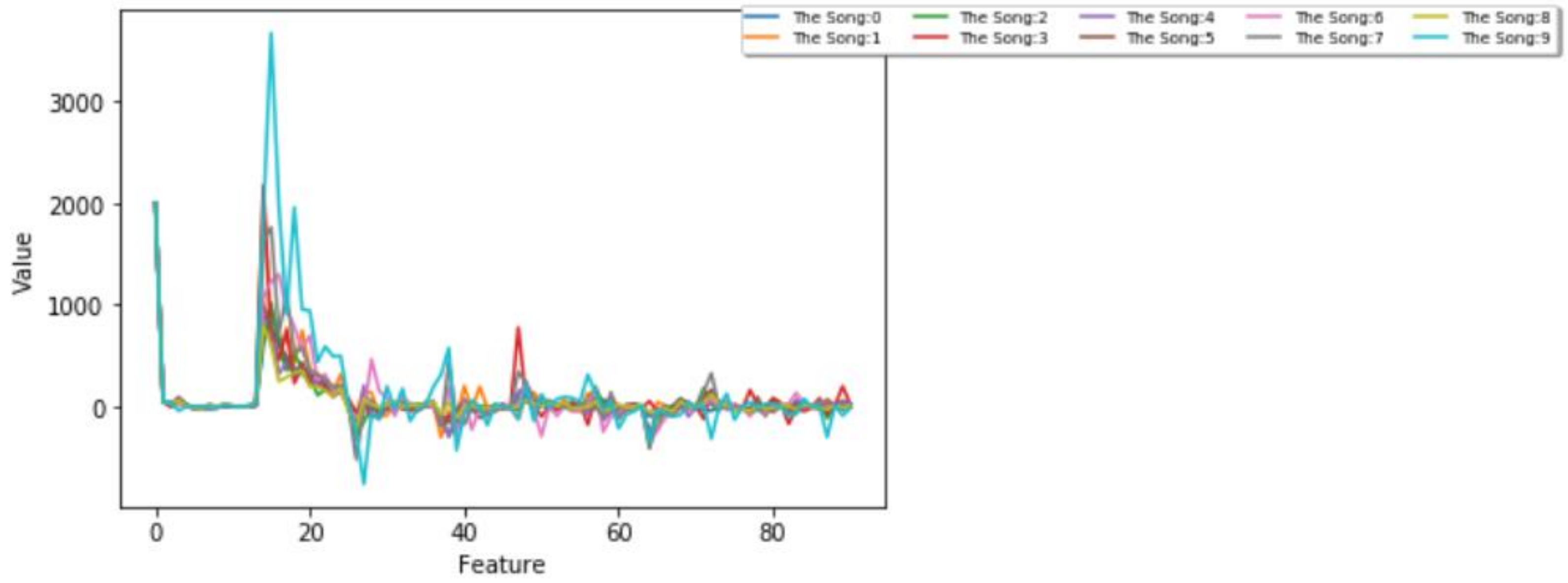
- Can find it is imbalance. From 1922 to 2007, the number of songs increased by years, reach peak in 2007
- Reduce from 2008 to 2011
- It may be related to the trend of music industry

Data Values Distribution Plot



- Show first 12 columns which are timbre average
- Can see the range is from minus 300 to 400
- Most of the data values near to 0

The First 10 Samples Plot



- Mostly samples have high values between column 12 and 20
- Have lowest values between column 22 and 28

Skewness

	summary	label
0	count	515345
1	mean	1998.3970815667174
2	stddev	10.931046354331716
3	min	1922
4	max	2011
5	25%	1994
6	50%	2002
7	75%	2006

```
+------------------+
|  skewness of label|
+------------------+
|-1.7315448701056186|
+------------------+
```

- Five number summary of the target
- Distribution is left skewed

Outline

Introduction
- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion
- Summary
- Model Comparation
- Lessons Learned

Spark MLlib Modeling

- Decision Tree Regressor
- Random Forest Regressor
- Decision Tree Regressor

Every model are evaluated with MSE (Mean Squared Error)

Evaluation Method

- We determine which model performs best by comparing which model has the best MSE results on the validation set.

- In the regression model, these two types of verification can determine the difference between the predicted data and the original data.

$$MSE = \frac{1}{N}\sum_{t=1}^{N}(observed_t - predicted_t)^2$$

Data Preprocess For Pyspark

- The dataset has one label and 90 non-label attributes
- With the 90 non- lebal attributes are assembled using Vector Assembler.

```
In [6]: from pyspark.ml.feature import VectorAssembler
        dataT_assembler = VectorAssembler(inputCols=['ta1','ta2','ta3','ta4','ta5','ta6','ta7',\
                                'ta8','ta9','ta10','ta11','ta12',\
                                't1','t2','t3','t4',\
                                't5','t6','t7','t8','t9','t10','t11','t12','t13','t14',\
                                't15','t16','t17','t18','t19','t20','t21','t22','t23',\
                                't24','t25','t26','t27','t28','t29','t30','t31','t32',\
                                't33','t34','t35','t36','t37','t38','t39','t40','t41',\
                                't42','t43','t44','t45','t46','t47','t48','t49','t50',\
                                't51','t52','t53','t54','t55','t56','t57','t58','t59',\
                                't60','t61','t62','t63','t64','t65','t66','t67','t68',\
                                't69','t70','t71','t72','t73','t74','t75','t76','t77',\
                                't78'], outputCol='features')
```

```
In [7]: dataT = dataT_assembler.transform(dataT)
```

```
In [8]: # model set with the features and label
        model_dataT=dataT.select(['features','label'])
```

Data Split For Pyspark

- By using limit() and sort(), the sample data is successfully segmented into a training set with 463,715 data volumes and a test set with 51,630 data volumes.

In [13]:

```
# got the train data with the first 463715
tr = model_dataT.limit(463715)

# got the test data with the last 51630
# Add an 'index' attribute as a standard for sorting
from pyspark.sql.functions import monotonically_increasing_id
t0 = model_dataT.withColumn("index",monotonically_increasing_id())
#t0.count()
# Convert to reverse
t1 = t0.sort("index",ascending = False)
#t1.count()
# select the first 51630 in the reverse (which means the last 51630 in original data)
t2 = t1.limit(51630)
#t2.count()
#Conversion order as positive sequence
t3 = t2.sort("index",ascending = True)
#t3.count()
#Remove 'index' back to the original data composition.
t4 = t3.drop('index')
#t4.show()
```

Decision Tree Regressor Model

- For models of decision tree regression, most attributes are set to default values, such as impurity (information gain calculation criteria), minInfoGain (minimum information gain required to split nodes). We decided to adjust maxBins and maxDepth to get the smallest possible MSE.

- When the parameter of the largest category is set to 5 in the Vector Indexer, the whole model has better results.

```
dt = DecisionTreeRegressor(featuresCol="indexedFeatures",maxDepth = 10,maxBins = 32)
```

- As a result, when maxBins is 32 and maxDepth is 10, the decision tree can get the minimum MSE when the maximum category is set to 5, and the value is 94.3053.

Random Forest Regressor Model

- Because the results of decision tree regression are not good enough, and it is impossible to determine whether there is an impact of outliers, the choice of multiple forests such as random forests reduces the impact of single decision trees caused by outliers. Inaccurate and avoid the possibility of overfitting.

- However, random forests also have core shortcomings, and the calculation of comparisons is particularly large.

```
#rf = RandomForestRegressor(featuresCol="indexedFeatures",maxDepth = 8 ,maxBins = 32)
rf = RandomForestRegressor(featuresCol="indexedFeatures",maxDepth = 10 ,maxBins = 32)
```

- As a result, when maxBins is 32 and maxDepth is 10, the decision tree can get the minimum MSE when the maximum category is set to 5, and the value is 94.3053.

Linear Regression Model

- Use linear regression to predict this data in pyspark. Two different regularizations are used to determine the range of the approximate data.

- Use elasticNetParam in linear regression functions to aid in regularization. When elasticNetParam is 1, the model uses Lasso regression. When 0, the model uses redge regression. At the same time, the maximum number of iterations is controlled to 30 times. The regularization parameter is 0.3.

```
#L1
lr = LinearRegression(labelCol="label",featuresCol="features", maxIter=30, regParam=0.3, elasticNetParam=1)
#L2
#lr = LinearRegression(labelCol="label",featuresCol="features", maxIter=30, regParam=0.3, elasticNetParam=0)
```

- The MSE range for the final result is between 90.015526 and 94.589867 (the first is L2 and the last is L1)

Interesting Finding For Spark MLlib

- Comparing the three models, the random forest has the best results in pyspark.

- For linear regression, the amount of data for this data is large, so the final prediction results are unsatisfactory. But it also shows that the data set has a certain linear relationship.

- More interesting is the maximum discrete feature number of the continuous feature discretization in the tree model is 32, and this value is exactly the default parameter of the property.

- On the parameter adjustment of the maximum depth, when the maximum tree depth exceeds 10, the final calculated MSE will increase. It can be seen that parameter 10 is similar to the valley of the quadratic function in the tree model of the data. Take this value and get the smallest value in different tree models

Outline

Introduction
- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion
- Summary
- Model Comparation
- Lessons Learned

Tensorflow-Keras Modeling

- Multi-layer Perceptron
- Convolutional Neural Network
- Recursive Neural Network

Every model are evaluated with MSE (Mean Squared Error)

Loading Data And Preprocess

```
x_train = x[:463715]
x_test = x[463715:]
y_train = y[:463715]
y_test = y[463715:]
```

```
x = df.iloc[:, 1:].to_numpy()
y = df.iloc[:, 0].to_numpy()
```

```
x_train.shape
```

```
(463715, 90)
```

```
df.isnull().sum()
```

```
0      0
1      0
2      0
3      0
```

About Tensorflow And Keras

- Open-source by Google, to implement ML particularly NN. Input as multidimensional array, aka tensor. Models are represented by flowchart to perform on the input.

- Keras is high-level API, user-friendly

```
In [13]: opt = optimizers.Adam(lr=0.1)
```

Multi-layer Perceptron Model

```
In [14]: mlp_model = Sequential()

         mlp_model.add(Dense(64, input_dim=90, \
                             kernel_regularizer=\
                             regularizers.l2(0.01),\
                             activation='relu'))
         mlp_model.add(Dropout(0.2))
         mlp_model.add(Dense(32, activation='relu'))
         mlp_model.add(Dropout(0.2))
         mlp_model.add(Dense(16, activation='relu'))
         mlp_model.add(Dropout(0.2))
         mlp_model.add(Dense(8, activation='relu'))
         mlp_model.add(Dropout(0.2))
         mlp_model.add(Dense(1, activation='linear'))
```

Multi-layer Perceptron Model Continue

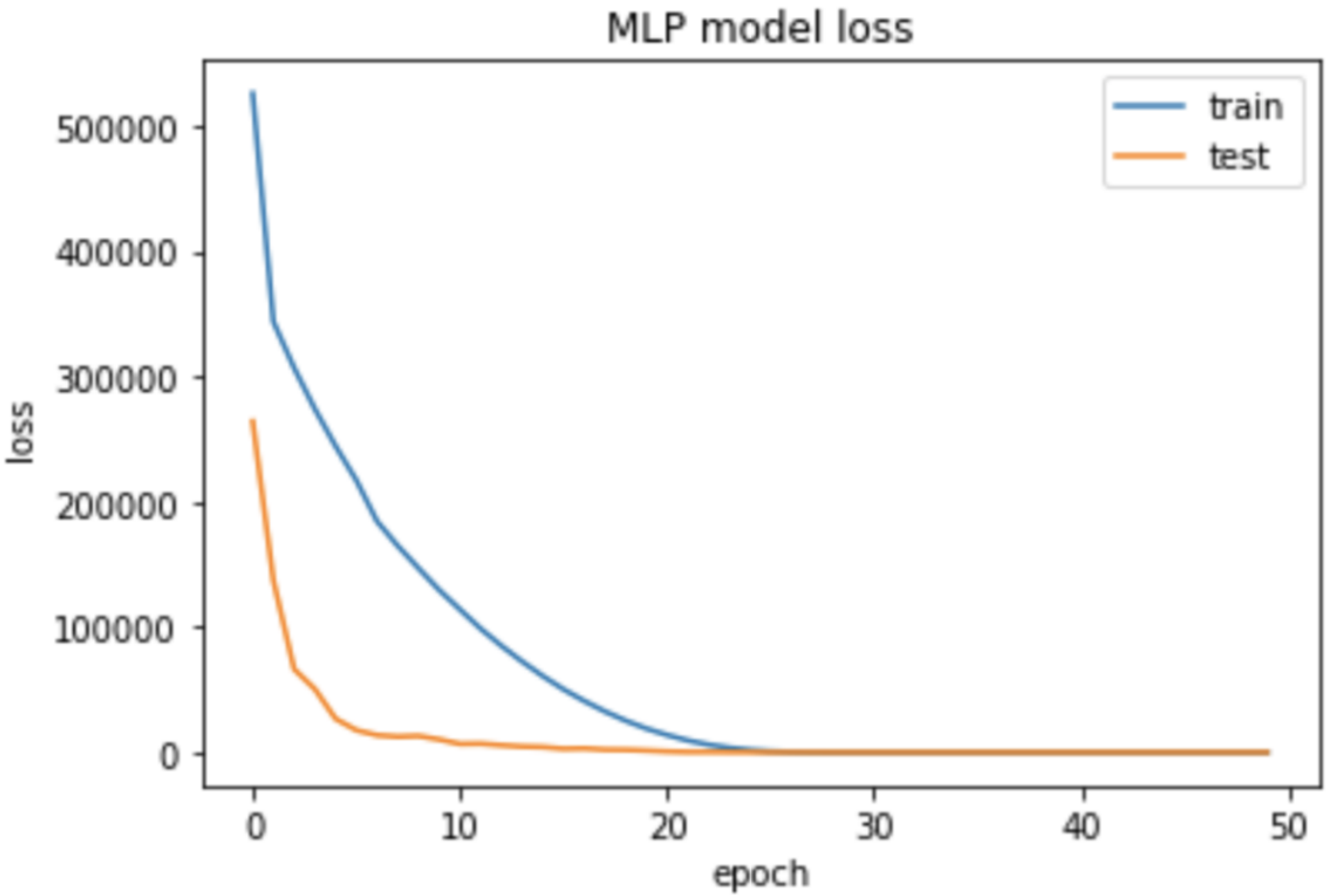
```
mlp_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                5824
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_3 (Dense)              (None, 16)                528
_________________________________________________________________
dropout_3 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_4 (Dense)              (None, 8)                 136
_________________________________________________________________
dropout_4 (Dropout)          (None, 8)                 0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 9
=================================================================
Total params: 8,577
Trainable params: 8,577
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer Perceptron Model Continue

```
Epoch 46/50
 - 3s - loss: 119.6996 - mse: 119.6996 - val_loss: 117.8360 - val_mse: 117.8360
Epoch 47/50
 - 3s - loss: 119.7061 - mse: 119.7061 - val_loss: 117.8077 - val_mse: 117.8077
Epoch 48/50
 - 3s - loss: 119.7053 - mse: 119.7054 - val_loss: 117.8844 - val_mse: 117.8844
Epoch 49/50
 - 3s - loss: 119.7068 - mse: 119.7068 - val_loss: 117.7734 - val_mse: 117.7734
Epoch 50/50
 - 3s - loss: 119.7014 - mse: 119.7014 - val_loss: 117.7666 - val_mse: 117.7666
```

Multi-layer Perceptron Model Continue



Convolutional Neural Network Model

```
x_train = x_train.reshape(463715, 90, 1)
x_test = x_test.reshape(51630, 90, 1)
```

```
x_train.shape
```

```
(463715, 90, 1)
```

```
cnn_model.add(Conv1D(filters=32, kernel_size=5, \
                     input_shape=(90,1), \
                     activation='relu'))
cnn_model.add(MaxPool1D(pool_size=3))
cnn_model.add(Flatten())
cnn_model.add(Dense(128, activation = 'relu'))
cnn_model.add(Dense(64, activation = 'relu'))
cnn_model.add(Dense(1, activation = 'linear'))
```

Convolutional Neural Network Model Continue

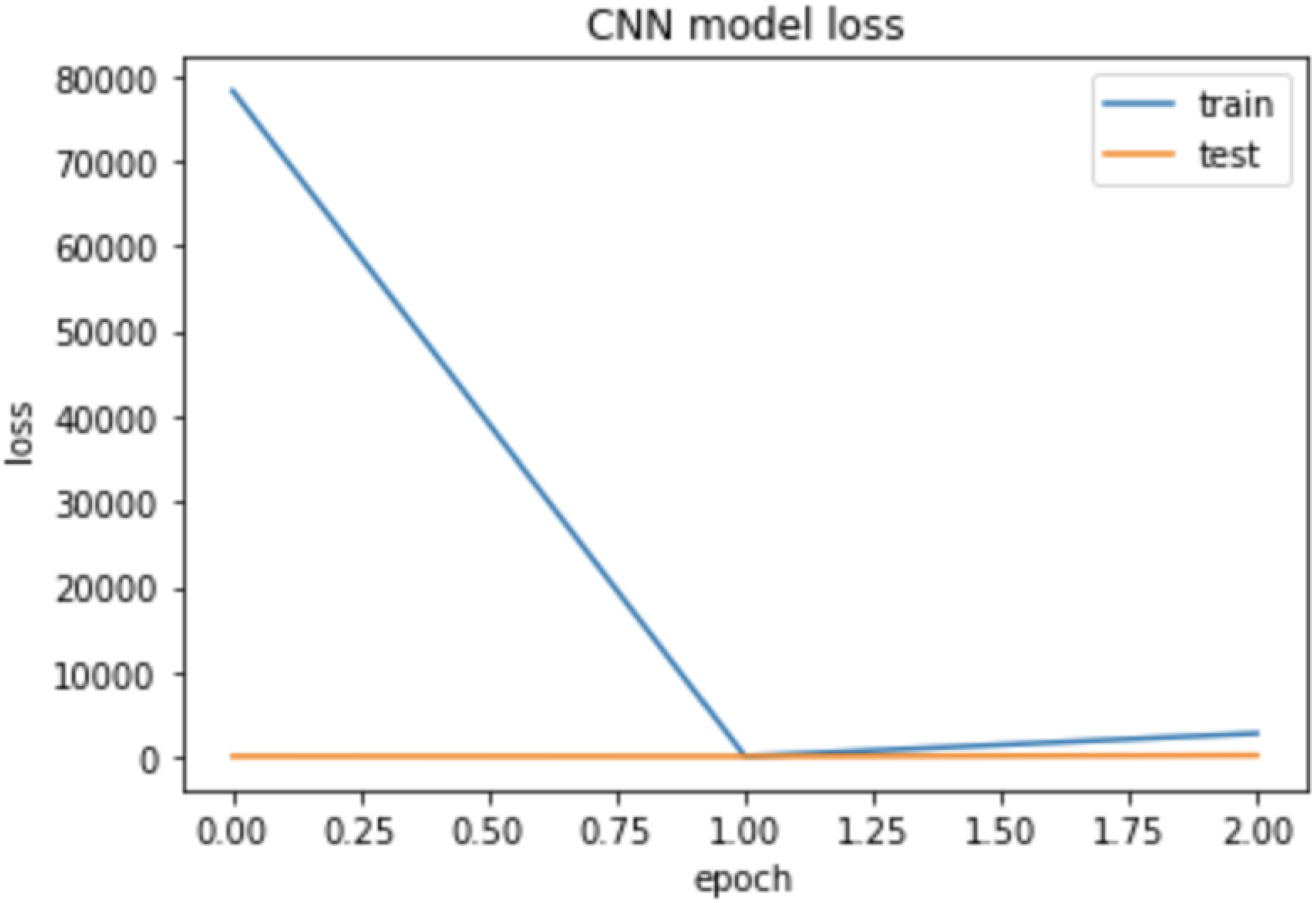
```
cnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 86, 32)            192
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 28, 32)            0
_________________________________________________________________
flatten_1 (Flatten)          (None, 896)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               114816
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 123,329
Trainable params: 123,329
Non-trainable params: 0
_________________________________________________________________
```

Convolutional Neural Network Model Continue

```
Train on 463715 samples, validate on 51630 samples
Epoch 1/3
 - 15s - loss: 78194.2009 - mse: 78194.1719 - val_loss: 150.4983 - val
_mse: 150.4983
Epoch 2/3
 - 14s - loss: 163.1502 - mse: 163.1501 - val_loss: 138.8521 - val_mse
: 138.8521
Epoch 3/3
 - 14s - loss: 2831.5630 - mse: 2831.5642 - val_loss: 261.1135 - val_m
se: 261.1136
```

Convolutional Neural Network Model Continue



Recursive Neural Network Model

In [13]:
```
rnn_model = Sequential()
rnn_model.add(Embedding(90, 128, input_length=90))
rnn_model.add(Bidirectional(LSTM(64)))
rnn_model.add(Dropout(0.5))
rnn_model.add(Dense(1, activation='relu'))
```

In [15]:
```
rnn_model.summary()
```

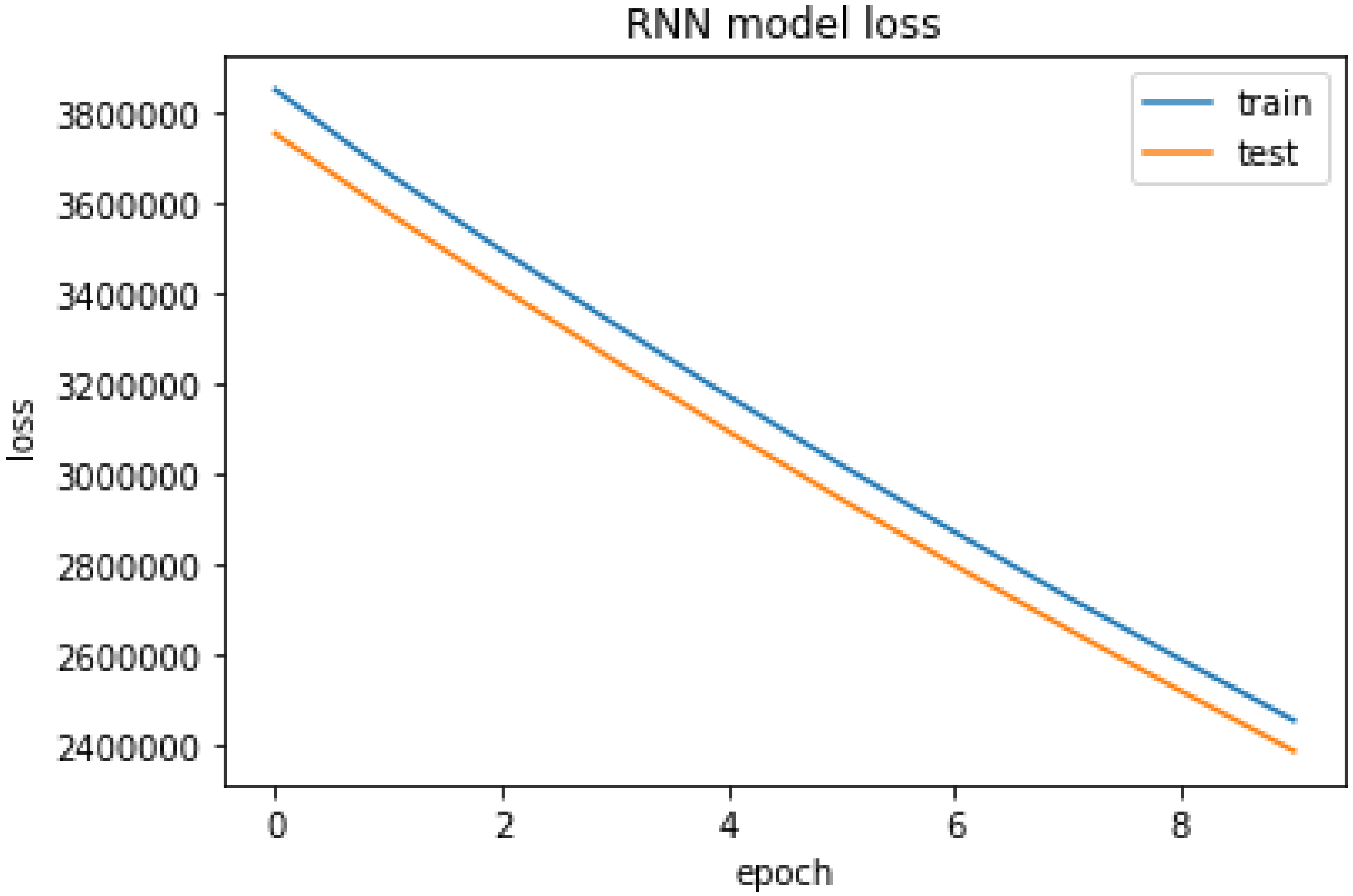
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 90, 128)           11520
_________________________________________________________________
bidirectional (Bidirectional (None, 128)               98816
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 110,465
Trainable params: 110,465
Non-trainable params: 0
_________________________________________________________________
```

Recursive Neural Network Model Continue

```
In [16]: history = rnn_model.fit(x_train, y_train, epochs=10, batch_size=1000, validation_data = [x_test, y_test])

Train on 360741 samples, validate on 154604 samples
WARNING:tensorflow:From C:\Users\hp\Anaconda3\lib\site-packages\tensorflow\python\ops\math_grad.py:1250: add_dispatch_support.<
locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Epoch 1/10
360741/360741 [==============================] - 747s 2ms/sample - loss: 3853539.5779 - mean_squared_error: 3853541.0000 - val_
loss: 3755579.7630 - val_mean_squared_error: 3755580.2500
Epoch 2/10
360741/360741 [==============================] - 779s 2ms/sample - loss: 3667422.1826 - mean_squared_error: 3667423.2500 - val_
loss: 3580329.6216 - val_mean_squared_error: 3580329.0000
Epoch 3/10
360741/360741 [==============================] - 768s 2ms/sample - loss: 3496403.0544 - mean_squared_error: 3496403.2500 - val_
loss: 3412790.6133 - val_mean_squared_error: 3412791.0000
Epoch 4/10
360741/360741 [==============================] - 772s 2ms/sample - loss: 3332164.2674 - mean_squared_error: 3332163.2500 - val_
loss: 3251314.1348 - val_mean_squared_error: 3251314.0000
Epoch 5/10
360741/360741 [==============================] - 785s 2ms/sample - loss: 3173531.9813 - mean_squared_error: 3173531.5000 - val_
loss: 3095218.8666 - val_mean_squared_error: 3095218.2500
Epoch 6/10
360741/360741 [==============================] - 771s 2ms/sample - loss: 3019756.4740 - mean_squared_error: 3019756.2500 - val_
loss: 2944092.3560 - val_mean_squared_error: 2944092.0000
Epoch 7/10
360741/360741 [==============================] - 713s 2ms/sample - loss: 2871243.7047 - mean_squared_error: 2871242.5000 - val_
loss: 2797709.0122 - val_mean_squared_error: 2797708.7500
Epoch 8/10
360741/360741 [==============================] - 708s 2ms/sample - loss: 2727106.4008 - mean_squared_error: 2727105.7500 - val_
loss: 2655838.2699 - val_mean_squared_error: 2655838.2500
Epoch 9/10
360741/360741 [==============================] - 708s 2ms/sample - loss: 2588386.6295 - mean_squared_error: 2588385.5000 - val_
loss: 2518385.2878 - val_mean_squared_error: 2518384.7500
Epoch 10/10
360741/360741 [==============================] - 708s 2ms/sample - loss: 2452872.1984 - mean_squared_error: 2452872.0000 - val_
loss: 2385119.4812 - val_mean_squared_error: 2385120.0000
```

Recursive Neural Network Model Continue



Interesting Finding For Tensorflow-Keras

- MLP most consistent, but slower than CNN
- CNN is very quick and can yield slightly better result but tends to overfit and is inconsistent
- RNN takes very long to compute and is infeasible for this type of problem

Outline

Self-Implemented Modeling

- Linear Regression
- Random Forest Regressor
- Decision Tree Regressor

Every model are evaluated with MSE (Mean Squared Error)

Linear Regression Model

Library we use in Linear Regression

- Numpy to do all of the vectorized numerical computations on the dataset including the implementation of the algorithm
- Matplotlib to plot graphs for better understanding the problem at hand with some visual aid
- Pandas to load the csv file into a dataframe format which is easier to plot.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

In [4]:
```
dataset = pd.read_csv('/Users/hp/Desktop/group/YearPredictionMSD.txt', header = None)
dataset
```

Linear Regression Model Continue

In [17]:
```
class LinearRegression:
    def __init__(self, X, y, alpha=0.03, n_iter=1500):
        self.alpha = alpha
        self.n_iter = n_iter
        self.n_samples = len(y)
        self.n_features = np.size(X, 1)
        self.X = np.hstack((np.ones((self.n_samples, 1)), (X - np.mean(X, 0)) / np.std(X, 0)))
        self.y = y[:, np.newaxis]
        self.params = np.zeros((self.n_features + 1, 1))
        self.coef_ = None
        self.intercept_ = None

    def fit(self):
        for i in range(self.n_iter):
            self.params = self.params - (self.alpha/self.n_samples) * \
            self.X.T @ (self.X @ self.params - self.y)
            self.intercept_ = self.params[0]
            self.coef_ = self.params[1:]
        return self

    def score(self, X=None, y=None):
        if X is None:
            X = self.X
        else:
            n_samples = np.size(X, 0)
            X = np.hstack((np.ones(
                (n_samples, 1)), (X - np.mean(X, 0)) / np.std(X, 0)))
        if y is None:
            y = self.y
        else:
            y = y[:, np.newaxis]
        y_pred = X @ self.params
        score = 1 - (((y - y_pred)**2).sum() / ((y - y.mean())**2).sum())
        return score

    def predict(self, X):
        n_samples = np.size(X, 0)
        y = np.hstack((np.ones((n_samples, 1)), (X-np.mean(X, 0)) / np.std(X, 0))) @ self.params
        return y

    def get_params(self):
        return self.params
```

In [36]:
```
X = dataset.iloc[:, 1:].as_matrix()
y = dataset.iloc[:, 0].as_matrix()
```

In [37]:
```
X_train = X[:463715]
X_test = X[463715:]
y_train = y[:463715]
y_test = y[463715:]
```

In [38]:
```
linReg = LinearRegression(X_train, y_train).fit()
```

In [39]:
```
preds = linReg.predict(X_test)
```

Linear Regression Model MSE And RMSE

Linear Regression Model MSE

In [40]:
```
from sklearn.metrics import mean_squared_error
def mse(h, y):
    return mean_squared_error(h, y)
mse(y_test, preds)
```

Out[40]: 90.49301918455576

Linear Regression Model RMSE

In [41]:
```
from math import sqrt
def rmse(h, y):
    return sqrt(mean_squared_error(h, y))
rmse(y_test, preds)
```

Out[41]: 9.512781884630582

Random Forest Model

Library we use in Random Forest

- Numpy to do all of the vectorized numerical computations on the dataset including the implementation of the algorithm
- Matplotlib to plot graphs for better understanding the problem at hand with some visual aid
- Pandas to load the csv file into a dataframe format which is easier to plot.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

In [4]:
```
dataset = pd.read_csv('/Users/hp/Desktop/group/YearPredictionMSD.txt', header = None)
dataset
```

Random Forest Model Continue

In [14]:
```
class DecisionTree():
    def __init__(self, x, y, idxs, min_leaf=5):
        self.x,self.y,self.idxs,self.min_leaf = x,y,idxs,min_leaf
        self.n,self.c = len(idxs), x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()

    def find_varsplit(self):
        for i in range(self.c): self.find_better_split(i)
        if self.score == float('inf'): return
        x = self.split_col
        lhs = np.nonzero(x<=self.split)[0]
        rhs = np.nonzero(x>self.split)[0]
        self.lhs = DecisionTree(self.x, self.y, self.idxs[lhs])
        self.rhs = DecisionTree(self.x, self.y, self.idxs[rhs])

    def find_better_split(self, var_idx):
        x,y = self.x.values[self.idxs,var_idx], self.y[self.idxs]
        sort_idx = np.argsort(x)
        sort_y,sort_x = y[sort_idx], x[sort_idx]
        rhs_cnt,rhs_sum,rhs_sum2 = self.n, sort_y.sum(), (sort_y**2).sum()
        lhs_cnt,lhs_sum,lhs_sum2 = 0,0.,0.

        for i in range(0,self.n-self.min_leaf):
            xi,yi = sort_x[i],sort_y[i]
            lhs_cnt += 1; rhs_cnt -= 1
            lhs_sum += yi; rhs_sum -= yi
            lhs_sum2 += yi**2; rhs_sum2 -= yi**2
            if i<self.min_leaf-1 or xi==sort_x[i+1]:
                continue

            lhs_std = std_agg(lhs_cnt, lhs_sum, lhs_sum2)
            rhs_std = std_agg(rhs_cnt, rhs_sum, rhs_sum2)
            curr_score = lhs_std*lhs_cnt + rhs_std*rhs_cnt
            if curr_score<self.score:
                self.var_idx,self.score,self.split = var_idx,curr_score,xi

    @property
    def split_name(self): return self.x.columns[self.var_idx]

    @property
    def split_col(self): return self.x.values[self.idxs,self.var_idx]

    @property
    def is_leaf(self): return self.score == float('inf')

    def __repr__(self):
        s = f'n: {self.n}; val:{self.val}'
        if not self.is_leaf:
            s += f'; score:{self.score}; split:{self.split}; var:{self.split_name}'
        return s

    def predict(self, x):
        return np.array([self.predict_row(xi) for xi in x])

    def predict_row(self, xi):
        if self.is_leaf: return self.val
        t = self.lhs if xi[self.var_idx]<=self.split else self.rhs
        return t.predict_row(xi)
```

In [10]:
```
x_train = x[:463715]
x_test = x[463715:]
y_train = y[:463715]
y_test = y[463715:]
```

In [7]:
```
x = df.iloc[:, 1:]
y = df.iloc[:, 0]
```

In [13]:
```
class RandomForest():
    def __init__(self, x, y, n_trees, sample_sz, min_leaf=5):
        np.random.seed(42)
        self.x,self.y,self.sample_sz,self.min_leaf = x,y,sample_sz,min_leaf
        self.trees = [self.create_tree() for i in range(n_trees)]

    def create_tree(self):
        idxs = np.random.permutation(len(self.y))[:self.sample_sz]
        return DecisionTree(self.x.iloc[idxs], self.y[idxs],
                    idxs=np.array(range(self.sample_sz)), min_leaf=self.min_leaf)

    def predict(self, x):
        return np.mean([t.predict(x) for t in self.trees], axis=0)

def std_agg(cnt, s1, s2): return math.sqrt((s2/cnt) - (s1/cnt)**2)
```

Random Forest Model MSE And RMSE

Random Forest Model MSE

```
In [16]: mse(y_test, preds)
Out[16]: 117.8514197172186З
```

Random Forest Model RMSE

```
In [17]: rmse(y_test, preds)
Out[17]: 10.855939375163194
```

Decision Tree Model

Libraries we used in Decision Tree

- Numpy to do all of the vectorized numerical computations on the dataset including the implementation of the algorithm
- Matplotlib to plot graphs for better understanding the problem at hand with some visual aid
- Pandas to load the csv file into a dataframe format which is easier to plot.

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
```

```
In [4]: dataset = pd.read_csv('/Users/hp/Desktop/group/YearPredictionMSD.txt', header = None)
        dataset
```

Decision Tree Model Continue

In [23]:
```python
class Node:

    def __init__(self, x, y, idxs, min_leaf=5):
        self.x = x
        self.y = y
        self.idxs = idxs
        self.min_leaf = min_leaf
        self.row_count = len(idxs)
        self.col_count = x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()

    def find_varsplit(self):
        for c in range(self.col_count): self.find_better_split(c)
        if self.is_leaf: return
        x = self.split_col
        lhs = np.nonzero(x <= self.split)[0]
        rhs = np.nonzero(x > self.split)[0]
        self.lhs = Node(self.x, self.y, self.idxs[lhs], self.min_leaf)
        self.rhs = Node(self.x, self.y, self.idxs[rhs], self.min_leaf)

    def find_better_split(self, var_idx):

        x = self.x.values[self.idxs, var_idx]

        for r in range(self.row_count):
            lhs = x <= x[r]
            rhs = x > x[r]
            if rhs.sum() < self.min_leaf or lhs.sum() < self.min_leaf: continue

            curr_score = self.find_score(lhs, rhs)
            if curr_score < self.score:
                self.var_idx = var_idx
                self.score = curr_score
                self.split = x[r]

    def find_score(self, lhs, rhs):
        y = self.y[self.idxs]
        lhs_std = y[lhs].std()
        rhs_std = y[rhs].std()
        return lhs_std * lhs.sum() + rhs_std * rhs.sum()

    @property
    def split_col(self): return self.x.values[self.idxs,self.var_idx]

    @property
    def is_leaf(self): return self.score == float('inf')

    def predict(self, x):
        return np.array([self.predict_row(xi) for xi in x])

    def predict_row(self, xi):
        if self.is_leaf: return self.val
        node = self.lhs if xi[self.var_idx] <= self.split else self.rhs
        return node.predict_row(xi)
```

In [22]:
```python
class DecisionTreeRegressor:

    def fit(self, x_train, y_train, min_leaf = 5):
        self.dtree = Node(x_train, y_train, np.array(np.arange(len(y_train))), min_leaf)
        return self

    def predict(self, x_train):
        return self.dtree.predict(x_train.values)
```

In [24]:
```python
regressor = DecisionTreeRegressor().fit(x_train, y_train)
preds = regressor.predict(x_test)
```

Decision Tree Model MSE And RMSE

Decision Tree Model MSE

```
In [27]: def mse(h, y):
             return mean_squared_error(h, y)
         mse(y_test, preds)

Out[27]: 195.07073270204978
```

Decision Tree Model RMSE

```
In [26]: from sklearn.metrics import mean_squared_error
         rmse(y_test, preds)

Out[26]: 13.96677245114453
```

Interesting Finding For Self-Implemented Models

- Linear regression are more suitable for this kind of problem if we conclude based on MSE/RMSE only.
- Decision Tree used up too much memory to run and took a long time to finish.
- Random Forest is suitable but the MSE/RMSE is not better than Linear Regression

Outline

Introduction
- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion
- Summary
- Model Comparation
- Lessons Learned

Summary

- PySpark model of Random Forest have the best MSE from all 9 models.
- Keras library need to be fine tuned to achieve maximum result
- PySpark is a powerful machine learning library
- Although we have all these powerful machine learning library, self implemented machine learning model are still a plausible choice

Outline

Introduction
- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion
- Summary
- Model Comparation
- Lessons Learned

Model Comparation

Spark MLlib Models

	Linear Regression	Decision Tree Regressor	Random Forest Regressor
Mean square error	90.0155	94.3053	89.3555

Tensorflow-Keras Models

	Multi-layer Perceptron	Recursive Neural Network	Convolutional Neural Network
Mean square error	119.7014 `	2831.5642	2452872.1984

Self-Implemented Models

	Linear Regression	Decision Tree Regressor	Random Forest Regressor
Mean square error	90.4930	195.0707	117.8514

- For Spark MLlib Models Random Forest Regressor get better result
- For Tensorflow-Keras get better result
- For Self-Implemented Models Linear Regression get better result

Outline

Introduction

- Overview
- Task Flow

Explorative Analysis And Visualization

Spark Mllib Modeling

Tensorflow-Keras Modeling

Self-Implemented Modeling

Conclusion

- Summary
- Model Comparation
- Lessons Learned

Lessons Learned

- From our project we learn that Pyspark mllib is the most suitable approach for this dataset. Because it is fast reliable and have good MSE

- There are still possible ways to reduce MSE further by changing the hyperparameters of the models, such as the learning rate, optimizer, maxDepth

- Recursive Neural Network model is more suitable for image processing, not very suitable for the regression problem

Thank you for listening