```
import warnings
warnings.simplefilter('ignore')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import re
from time import time
from scipy import stats
import json

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.metrics import make_scorer, roc_auc_score, log_loss, accuracy_score
from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import confusion_matrix
from IPython.display import display, Math, Latex



from sklearn.linear_model import Lasso,Ridge,LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier
# import xgboost as xgb

# Read data from application_train dataset.
data = pd.read_csv('/content/drive/MyDrive/application_train.csv')
data.head()
```

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_(
0	100002	1	Cash loans	M	N	
1	100003	0	Cash loans	F	N	
2	100004	0	Revolving loans	M	Y	
3	100006	0	Cash loans	F	N	
4	100007	0	Cash loans	M	N	

5 rows × 122 columns

```
y = data['TARGET']
X = data.drop(['SK_ID_CURR','TARGET'], axis = 1)
```

Additional EDA

Discarding features with null values more than 30%

```
null_data = X.isna().sum().reset_index().rename(columns={'index':'col_name',0:'null_c
null_data['count_%'] = null_data['null_count']/len(X)*100
null_data = null_data[null_data['count_%'] <= 30]
null_data
```

	col_name	null_count	count_%
0	NAME_CONTRACT_TYPE	0	0.000000
1	CODE_GENDER	0	0.000000

```
selected_columns = null_data['col_name'].tolist() + ['TARGET']
print(selected_columns)
```

```
['NAME_CONTRACT_TYPE', 'CODE_GENDER', 'FLAG_OWN_CAR', 'FLAG_OWN_REALTY', 'CNT_CH
```

```
null_data['col_type'] = null_data['col_name'].apply(lambda x: X[x].dtype)
null_data[null_data['count_%'] > 0]
```

	col_name	null_count	count_%	col_type
7	AMT_ANNUITY	12	0.003902	float64
8	AMT_GOODS_PRICE	278	0.090403	float64
9	NAME_TYPE_SUITE	1292	0.420148	object
27	CNT_FAM_MEMBERS	2	0.000650	float64
40	EXT_SOURCE_2	660	0.214626	float64
41	EXT_SOURCE_3	60965	19.825307	float64
89	OBS_30_CNT_SOCIAL_CIRCLE	1021	0.332021	float64
90	DEF_30_CNT_SOCIAL_CIRCLE	1021	0.332021	float64
91	OBS_60_CNT_SOCIAL_CIRCLE	1021	0.332021	float64
92	DEF_60_CNT_SOCIAL_CIRCLE	1021	0.332021	float64
93	DAYS_LAST_PHONE_CHANGE	1	0.000325	float64
114	AMT_REQ_CREDIT_BUREAU_HOUR	41519	13.501631	float64
115	AMT_REQ_CREDIT_BUREAU_DAY	41519	13.501631	float64
116	AMT_REQ_CREDIT_BUREAU_WEEK	41519	13.501631	float64
117	AMT_REQ_CREDIT_BUREAU_MON	41519	13.501631	float64
118	AMT_REQ_CREDIT_BUREAU_QRT	41519	13.501631	float64
119	AMT_REQ_CREDIT_BUREAU_YEAR	41519	13.501631	float64

Filling null values in NAME_TYPE_SUITE column with "Other_C"

```
X_feature = data[selected_columns]
X_feature['NAME_TYPE_SUITE'].fillna('Other_C', inplace=True)
```

```
X_feature.head()
```

	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHIL
0	Cash loans	M	N	Y	
1	Cash loans	F	N	N	
2	Revolving loans	M	Y	Y	
3	Cash loans	F	N	Y	
4	Cash loans	M	N	Y	

Filling null values in the columns containing keyword AMT_REQ_CREDIT column with 0

```
temp_col_reqd = null_data[null_data['null_count'] != 0].reset_index(drop=True)['col_n
for col in temp_col_reqd:
    if 'AMT_REQ_CREDIT' in col:
        print("columns to be filled with 0 is: {}".format(col))
        X_feature[col].fillna(0,inplace=True)
```

```
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_HOUR
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_DAY
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_WEEK
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_MON
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_QRT
columns to be filled with 0 is: AMT_REQ_CREDIT_BUREAU_YEAR
```

Filling null values in the column containing keyword CNT_SOCIAL_CIRCLE with 0

```
for col in temp_col_reqd:
    if 'CNT_SOCIAL_CIRCLE' in col:
        print("columns to be filled with 0 is: {}".format(col))
        X_feature[col].fillna(0,inplace=True)
```

```
columns to be filled with 0 is: OBS_30_CNT_SOCIAL_CIRCLE
columns to be filled with 0 is: DEF_30_CNT_SOCIAL_CIRCLE
columns to be filled with 0 is: OBS_60_CNT_SOCIAL_CIRCLE
columns to be filled with 0 is: DEF_60_CNT_SOCIAL_CIRCLE
```

Filling null values in the column CNT_FAM_MEMBERS with median

```
for col in temp_col_reqd:
    if 'CNT_FAM_MEMBERS' in col:
        print("columns to be filled with median is: {}".format(col))
        X_feature[col].fillna(X_feature[col].median(),inplace=True)
```

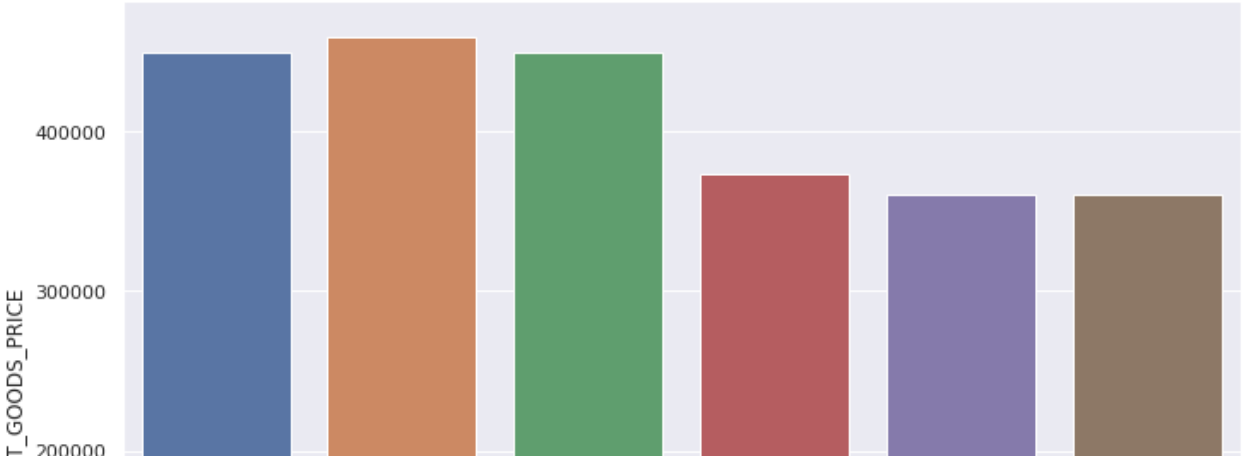
```
 columns to be filled with median is: CNT_FAM_MEMBERS
```

Filling null values in the column AMT_GOODS_PRICE with median fopr the respective category

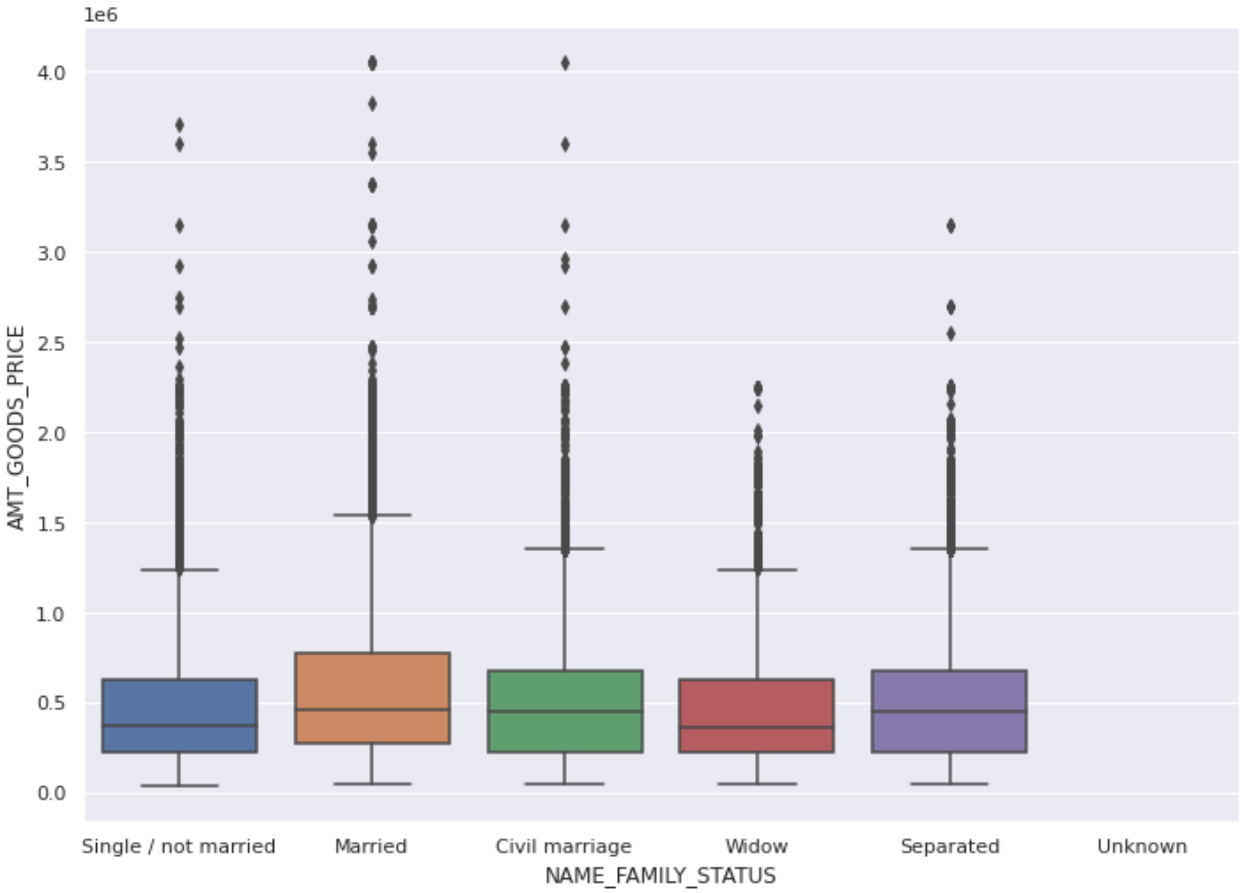
```
temp_plt_data = X_feature[['AMT_GOODS_PRICE','NAME_FAMILY_STATUS']]
temp_plt_data = temp_plt_data.groupby('NAME_FAMILY_STATUS')['AMT_GOODS_PRICE'].median
temp_plt_data['AMT_GOODS_PRICE'] = temp_plt_data['AMT_GOODS_PRICE'].fillna(temp_plt_d
temp_plt_data.head()
```

	NAME_FAMILY_STATUS	AMT_GOODS_PRICE
0	Civil marriage	450000.0
1	Married	459000.0
2	Separated	450000.0
3	Single / not married	373500.0
4	Unknown	360000.0

```
sns.set(rc={'figure.figsize':(11.7,8.27)})
ax = sns.barplot(x="NAME_FAMILY_STATUS", y="AMT_GOODS_PRICE", data=temp_plt_data)
```




```
sns.set(rc={'figure.figsize':(11.7,8.27)})
ax = sns.boxplot(x="NAME_FAMILY_STATUS", y="AMT_GOODS_PRICE", data=X_feature)
```




```
def fill_category_value(a):
    if a['AMT_GOODS_PRICE'] == np.inf:
        return temp_plt_data[temp_plt_data['NAME_FAMILY_STATUS']==a['NAME_FAMILY_STAT
    else:
        return a['AMT_GOODS_PRICE']
```

```
for col in temp_col_reqd:
    X_feature['AMT_GOODS_PRICE'] = X_feature['AMT_GOODS_PRICE'].fillna(np.inf)
    if 'AMT_GOODS_PRICE' in col:
        print("columns to be filled with category median is: {}".format(col))
        X_feature['AMT_GOODS_PRICE'] = X_feature.apply(lambda a: fill_category_value(
```

```
 columns to be filled with category median is: AMT_GOODS_PRICE
```

Dropping one single row with column DAYS_LAST_PHONE_CHANGE as null

```
X_feature.dropna(subset=['DAYS_LAST_PHONE_CHANGE'], inplace=True)
```

Dropping 12 rows with column AMT_ANNUITY as null

```
X_feature.dropna(subset=['AMT_ANNUITY'], inplace=True)
```

```
X_feature = X_feature.reset_index(drop=True)
```

Checking highest correlated features with External Source to replace the null values with

Check for EXT_SOURCE_2

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

temp_corr_df = pd.DataFrame()

for col in X_feature.columns.tolist():
    if X_feature[col].dtype == 'int':
        l = [col, X_feature['EXT_SOURCE_2'].corr(X_feature[col])]
    else:
        l = [col, X_feature['EXT_SOURCE_2'].corr(pd.DataFrame(LabelEncoder().fit_tran
    temp_corr_df = temp_corr_df.append(pd.Series(l),ignore_index=True)
temp_corr_df = temp_corr_df.rename(columns={0:'col_name',1:'correlation_with_EXT_2'})
temp_corr_df['correlation_with_EXT_2'] = abs(temp_corr_df['correlation_with_EXT_2'])
temp_corr_df.sort_values(by='correlation_with_EXT_2',ascending=False).head(6).tail(5)
```

	col_name	correlation_with_EXT_2
26	REGION_RATING_CLIENT	0.292903
27	REGION_RATING_CLIENT_W_CITY	0.288306
43	DAYS_LAST_PHONE_CHANGE	0.195766
5	AMT_INCOME_TOTAL	0.170547

REGION_RATING_CLIENT : Our rating of the region where our client lives

```
region_rating_grouped = X_feature.groupby('REGION_RATING_CLIENT')['EXT_SOURCE_2'].med


def fill_external_source2(a):
    if a['EXT_SOURCE_2'] == np.inf:
        return region_rating_grouped[region_rating_grouped['REGION_RATING_CLIENT']==a
    else:
        return a['EXT_SOURCE_2']

X_feature['EXT_SOURCE_2'] = X_feature['EXT_SOURCE_2'].fillna(np.inf)
X_feature['EXT_SOURCE_2'] = X_feature.apply(lambda a: fill_external_source2(a),axis=1
```

Check for EXT_SOURCE_3

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

temp_corr_df = pd.DataFrame()

for col in X_feature.columns.tolist():
    if X_feature[col].dtype == 'int':
        l = [col, X_feature['EXT_SOURCE_3'].corr(X_feature[col])]
    else:
        l = [col, X_feature['EXT_SOURCE_3'].corr(pd.DataFrame(LabelEncoder().fit_tran
    temp_corr_df = temp_corr_df.append(pd.Series(l),ignore_index=True)
temp_corr_df = temp_corr_df.rename(columns={0:'col_name',1:'correlation_with_EXT_3'})
temp_corr_df['correlation_with_EXT_3'] = abs(temp_corr_df['correlation_with_EXT_3'])
temp_corr_df = temp_corr_df.sort_values(by='correlation_with_EXT_3',ascending=False).
temp_corr_df
```

	col_name	correlation_with_EXT_3
15	DAYS_BIRTH	0.205474
70	TARGET	0.178929
18	DAYS_ID_PUBLISH	0.131598
20	FLAG_EMP_PHONE	0.115284
16	DAYS_EMPLOYED	0.113426
37	EXT_SOURCE_2	0.109728

```
ext_source_data = X_feature[temp_corr_df['col_name'].tolist()+['EXT_SOURCE_3']]

for col in ext_source_data.columns.tolist():
    if col != 'EXT_SOURCE_3':
        ext_source_data[col] = LabelEncoder().fit_transform(X_feature[[col]])

ext_source3_train = ext_source_data[ext_source_data['EXT_SOURCE_3'].notnull()]
ext_source3_test = ext_source_data[ext_source_data['EXT_SOURCE_3'].isnull()]
ext_source3_train.shape, ext_source3_test.shape
```

```
((246535, 10), (60963, 10))
```

```
exs3_y_train = ext_source3_train[['EXT_SOURCE_3']]
exs3_X_train = ext_source3_train.drop(columns=['EXT_SOURCE_3'])

exs3_X_test = ext_source3_test.drop(columns=['EXT_SOURCE_3'])


from sklearn.linear_model import LinearRegression

model = LinearRegression().fit(exs3_X_train, exs3_y_train)
y_pred_exs3 = model.predict(exs3_X_test)

exs3_output = exs3_X_test
exs3_output['exs3_y'] = y_pred_exs3
exs3_output
```

	DAYS_BIRTH	TARGET	DAYS_ID_PUBLISH	FLAG_EMP_PHONE	DAYS_EMPLOYED
1	8382	0	5876	1	11384
3	6142	0	3730	1	9533
4	5215	0	2709	1	9534
9	10676	0	2175	1	10553
14	10562	0	4111	1	12369
...	...	...	...	...	...

```
exs3_output = exs3_output.reset_index().rename(columns={'index':'index_to_be_updated'
for i in exs3_output['index_to_be_updated'].tolist():
    X_feature['EXT_SOURCE_3'].iloc[i] = exs3_output[exs3_output['index_to_be_updated'
```

307491	8442	0	5908	1	5318

Checking the null values in the Train dataset

```
new_null_data = X_feature.isna().sum().reset_index().rename(columns={'index':'col_nam
new_null_data['count_%'] = new_null_data['null_count']/len(X_feature)*100
new_null_data = new_null_data[new_null_data['count_%'] <= 30]
new_null_data['col_type'] = new_null_data['col_name'].apply(lambda x: X_feature[x].dt
new_null_data[new_null_data['count_%'] > 0]
```

	col_name	null_count	count_%	col_type

```
X_feature.isna().sum()
```

```
NAME_CONTRACT_TYPE            0
CODE_GENDER                   0
FLAG_OWN_CAR                  0
FLAG_OWN_REALTY               0
CNT_CHILDREN                  0
                             ..
AMT_REQ_CREDIT_BUREAU_WEEK    0
AMT_REQ_CREDIT_BUREAU_MON     0
AMT_REQ_CREDIT_BUREAU_QRT     0
AMT_REQ_CREDIT_BUREAU_YEAR    0
TARGET                        0
Length: 71, dtype: int64
```

Training and testing with the selected columns

Adding Additional relevant features felt

```
X_feature['AMT_CREDIT_TO_ANNUITY_RATIO'] = X_feature['AMT_CREDIT'] / X_feature['AMT_A
X_feature['Tot_EXTERNAL_SOURCE'] = X_feature['EXT_SOURCE_2'] + X_feature['EXT_SOURCE_
X_feature['Salary_to_credit'] = X_feature['AMT_INCOME_TOTAL']/X_feature['AMT_CREDIT']
X_feature['Annuity_to_salary_ratio'] = X_feature['AMT_ANNUITY']/X_feature['AMT_INCOME
```

```
X_feature['TARGET'].value_counts()
```

```
0    282673
1     24825
Name: TARGET, dtype: int64
```

```
X_dump = X_feature
X_dump.shape
```

```
(307498, 75)
```

```
X_dump.to_csv('/content/drive/MyDrive/processed_training_data.csv', index=False)
```

```
import warnings
warnings.simplefilter('ignore')

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import re
from time import time
from scipy import stats
import json

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
```

```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.metrics import make_scorer, roc_auc_score, log_loss, accuracy_score
from sklearn.preprocessing import LabelEncoder,MinMaxScaler

from sklearn.metrics import confusion_matrix
from IPython.display import display, Math, Latex


from sklearn.linear_model import Lasso,Ridge,LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier
```

Building Binary Classification MLP using Pytorch

References: https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/

```
scaler = MinMaxScaler()
```

```
# pytorch mlp for binary classification
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import BCELoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
```

```
from sklearn.metrics import make_scorer, roc_auc_score

from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path).head(10000)

        num_attribs = []
        cat_attribs = []

        for col in df.columns.tolist():
            if df[col].dtype in (['int','float']):
                num_attribs.append(col)
            else:
                cat_attribs.append(col)

        le_dict = {}
        for col in df.columns.tolist():
            if df[col].dtype == 'object':
                le = LabelEncoder()
                df[col] = df[col].fillna("NULL")
                df[col] = le.fit_transform(df[col])
                le_dict['le_{}'.format(col)] = le

        # store the inputs and outputs
        self.X = df.drop(columns=['TARGET']).values[:, :]
        self.X = scaler.fit_transform(self.X)
        self.y = df['TARGET'].values[:]

        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.25):
        # determine sizes
```

```
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])

# model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 35)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(35, 15)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer
        self.hidden3 = Linear(15, 5)
        kaiming_uniform_(self.hidden3.weight, nonlinearity='relu')
        self.act3 = ReLU()
        # third hidden layer and output
        self.hidden4 = Linear(5, 1)
        xavier_uniform_(self.hidden4.weight)
        self.act4 = Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
         # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        X = self.act3(X)
        # third hidden layer and output
        X = self.hidden4(X)
        X = self.act4(X)
        return X

# prepare the dataset
def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
```

```
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

# train the model
def train_model(train_dl, model):
    # define the optimization
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in range(200):
        # enumerate mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # plotting on tensorboard
            writer.add_scalar("Loss/train", loss, epoch)
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()

# evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc

# make a class prediction for one row of data
def predict_model(test_dl, model):
    temp_df = pd.DataFrame()
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
```

```
            yhat = model(inputs)
            # retrieve numpy array
            yhat = yhat.detach().numpy()
            actual = targets.numpy()
            actual = actual.reshape((len(actual), 1))
            # round to class values
            yhat = yhat.round()
            # store
            predictions.append(yhat)
            actuals.append(actual)
        predictions = predictions[0].reshape(len(predictions[0])).tolist()
        actuals = actuals[0].reshape(len(actuals[0])).tolist()
        temp_df['pred'] = predictions
        temp_df['actual'] = actuals
        return temp_df

    # prepare the data
    path = '/content/drive/MyDrive/processed_training_data.csv'
    train_dl, test_dl = prepare_data(path)
    print(len(train_dl.dataset), len(test_dl.dataset))
    # define the network
    model = MLP(74)
    # train the model
    train_model(train_dl, model)
    # getting test results
    output_df = predict_model(test_dl, model)
    # evaluate the model
    acc = evaluate_model(test_dl, model)
    print('Accuracy: %.3f' % acc)

    writer.flush()
    writer.close()
```

```
    7500 2500
    Accuracy: 0.882
```

Trying the same classification task with different iterations of epochs, hidden layers and number of neurons per layers

```
# pytorch mlp for binary classification
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

```
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import BCELoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_
from sklearn.metrics import make_scorer, roc_auc_score

from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path).head(50000)

        num_attribs = []
        cat_attribs = []

        for col in df.columns.tolist():
            if df[col].dtype in (['int','float']):
                num_attribs.append(col)
            else:
                cat_attribs.append(col)

        le_dict = {}
        for col in df.columns.tolist():
            if df[col].dtype == 'object':
                le = LabelEncoder()
                df[col] = df[col].fillna("NULL")
                df[col] = le.fit_transform(df[col])
                le_dict['le_{}'.format(col)] = le

        # store the inputs and outputs
        self.X = df.drop(columns=['TARGET']).values[:, :]
        self.X = scaler.fit_transform(self.X)
        self.y = df['TARGET'].values[:]

        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
```

```
    def __len__(self):
        return len(self.X)

    # # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.25):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])

# model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer and output
        self.hidden2 = Linear(10, 1)
        xavier_uniform_(self.hidden2.weight)
        self.act2 = Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
         # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        return X

# prepare the dataset
def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    print(dataset)
    # calculate split
    train, test = dataset.get_splits()
    print(train)
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl
```

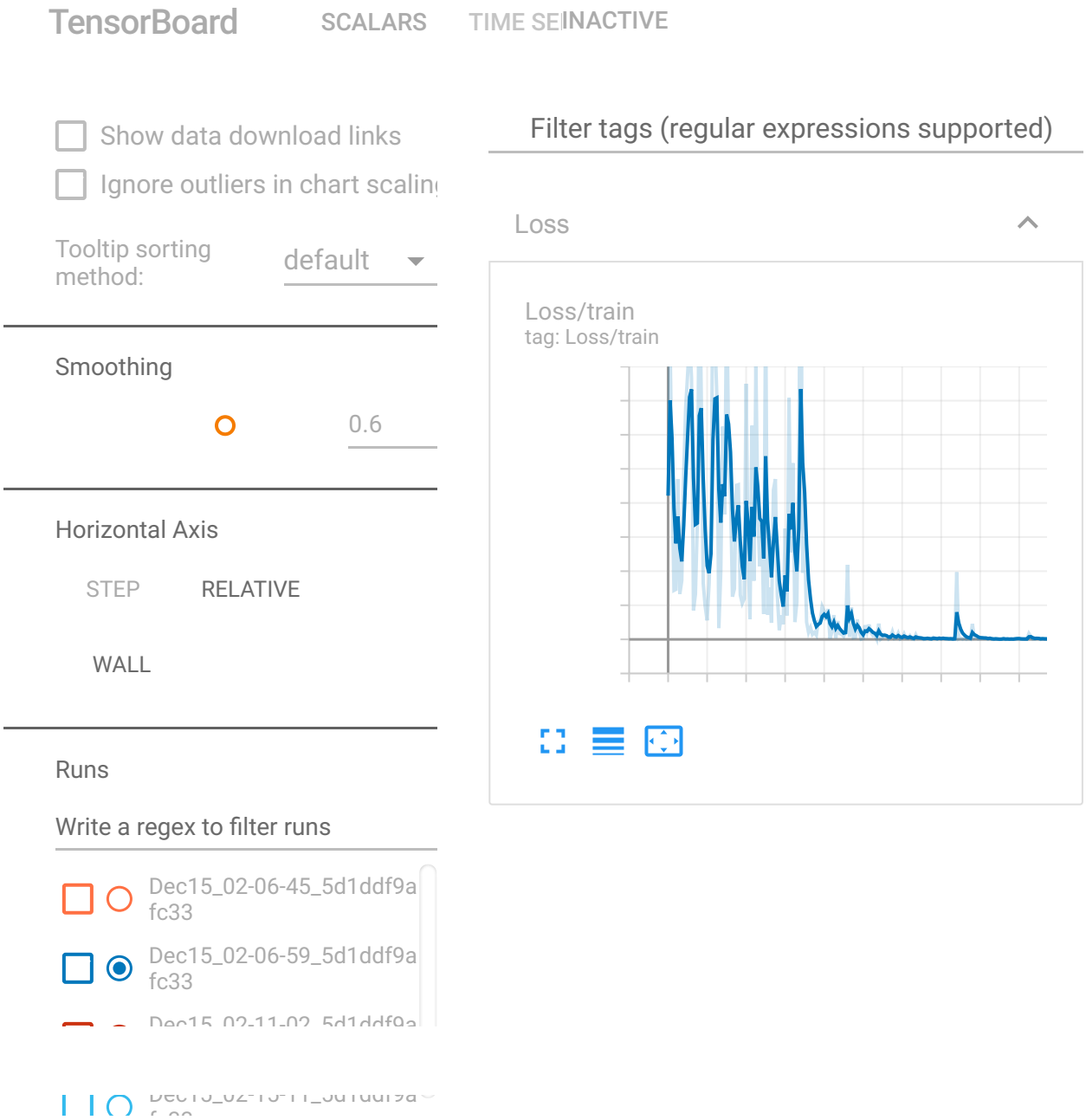
```
# train the model
def train_model(train_dl, model):
    # define the optimization
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in range(200):
        # enumerate mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # plotting on tensorboard
            writer.add_scalar("Loss/train", loss, epoch)
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()

# evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc

# make a class prediction for one row of data
def predict_model(test_dl, model):
    temp_df = pd.DataFrame()
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
```

```
            yhat = yhat.detach().numpy()
            actual = targets.numpy()
            actual = actual.reshape((len(actual), 1))
            # round to class values
            yhat = yhat.round()
            # store
            predictions.append(yhat)
            actuals.append(actual)
        predictions = predictions[0].reshape(len(predictions[0])).tolist()
        actuals = actuals[0].reshape(len(actuals[0])).tolist()
        temp_df['pred'] = predictions
        temp_df['actual'] = actuals
        return temp_df

    # prepare the data
    path = '/content/drive/MyDrive/processed_training_data.csv'
    train_dl, test_dl = prepare_data(path)
    print(len(train_dl.dataset), len(test_dl.dataset))
    # # define the network
    # model = MLP(74)
    # # train the model
    # train_model(train_dl, model)
    # # getting test results
    # output_df = predict_model(test_dl, model)
    # # evaluate the model
    # acc = evaluate_model(test_dl, model)
    # print('Accuracy: %.3f' % acc)

    # writer.flush()
    # writer.close()

    # AUC = roc_auc_score(output_df['actual'],output_df['pred'])
    # print (AUC)
```

```
    <__main__.CSVDataset object at 0x7f194a250050>
    <torch.utils.data.dataset.Subset object at 0x7f19497aaad0>
    37500 12500
```

```
    %load_ext tensorboard
    %tensorboard --logdir runs
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Reusing TensorBoard on port 6006 (pid 308), started 0:44:28 ago. (Use '!kil
```




```
test_dataset = pd.read_csv('/content/application_test.csv')
test_dataset = test_dataset[list(set(read_csv(path).columns.tolist())-set(['Annuity_t
test_dataset.head()
```

	AMT_INCOME_TOTAL	FLAG_DOCUMENT_16	FLAG_DOCUMENT_4	FLAG_PHONE	FLAG_DO
0	135000.0	0	0	0	
1	99000.0	0	0	0	

```
test_dataset['AMT_CREDIT_TO_ANNUITY_RATIO'] = test_dataset['AMT_CREDIT'] / test_datas
test_dataset['Tot_EXTERNAL_SOURCE'] = test_dataset['EXT_SOURCE_2'] + test_dataset['EX
test_dataset['Salary_to_credit'] = test_dataset['AMT_INCOME_TOTAL']/test_dataset['AMT
test_dataset['Annuity_to_salary_ratio'] = test_dataset['AMT_ANNUITY']/test_dataset['A
```

```
test_dataset.head()
```

	AMT_INCOME_TOTAL	FLAG_DOCUMENT_16	FLAG_DOCUMENT_4	FLAG_PHONE	FLAG_DO
0	135000.0	0	0	0	
1	99000.0	0	0	0	
2	202500.0	0	0	0	
3	315000.0	0	0	1	
4	180000.0	0	0	0	

```
test_dataset.shape
```

```
(48744, 74)
```

```
num_attribs = []
cat_attribs = []

for col in test_dataset.columns.tolist():
    if test_dataset[col].dtype in (['int','float']):
        num_attribs.append(col)
    else:
        cat_attribs.append(col)

le_dict = {}
for col in test_dataset.columns.tolist():
    if test_dataset[col].dtype == 'object':
        le = LabelEncoder()
        test_dataset[col] = test_dataset[col].fillna("NULL")
        test_dataset[col] = le.fit_transform(test_dataset[col])
        le_dict['le_{}'.format(col)] = le
# test_dataset = scaler.fit_transform(test_dataset)
```

```
test_dataset.head()
```

	AMT_INCOME_TOTAL	FLAG_DOCUMENT_16	FLAG_DOCUMENT_4	FLAG_PHONE	FLAG_DOCUMEN
0	135000.0	0	0	0	
1	99000.0	0	0	0	
2	202500.0	0	0	0	
3	315000.0	0	0	1	
4	180000.0	0	0	0	

```
test_dataset = test_dataset.values[:, :]
test_dataset = scaler.fit_transform(test_dataset)
test_data_dl = DataLoader(test_dataset, batch_size=32, shuffle=False)
test_data_dl
```

```
<torch.utils.data.dataloader.DataLoader at 0x7f1ad06f6fd0>
```

```
output_df = pd.DataFrame()
predictions, actuals = list(), list()
for i, (inputs, targets) in enumerate(test_data_dl):
    # evaluate the model on the test set
    yhat = model(inputs)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    actual = targets.numpy()
    actual = actual.reshape((len(actual), 1))
    # round to class values
    yhat = yhat.round()
    # store
    predictions.append(yhat)
    actuals.append(actual)
predictions = predictions[0].reshape(len(predictions[0])).tolist()
actuals = actuals[0].reshape(len(actuals[0])).tolist()
output_df['pred'] = predictions
output_df['actual'] = actuals
```

Build Regression MLP using pytorch

```
from numpy import vstack
from numpy import sqrt
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import MSELoss
from torch.nn.init import xavier_uniform_

# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path).head(1000)

        num_attribs = []
        cat_attribs = []

        for col in df.columns.tolist():
            if df[col].dtype in (['int','float']):
                num_attribs.append(col)
            else:
                cat_attribs.append(col)

        le_dict = {}
        for col in df.columns.tolist():
            if df[col].dtype == 'object':
                le = LabelEncoder()
                df[col] = df[col].fillna("NULL")
                df[col] = le.fit_transform(df[col])
                le_dict['le_{}'.format(col)] = le

        # store the inputs and outputs
        # self.X = df.drop(columns=['DAYS_EMPLOYED']).values[:, :]
        # self.y = df['DAYS_EMPLOYED'].values[:]
        self.X = df.drop(columns=['DAYS_EMPLOYED']).head(1000)
        self.y = df['DAYS_EMPLOYED'].head(1000)

        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
```

```
            self.y = self.y.reshape((len(self.y), 1))

        # number of rows in the dataset
        def __len__(self):
            return len(self.X)

        # get a row at an index
        def __getitem__(self, idx):
            return [self.X[idx], self.y[idx]]

        # get indexes for train and test rows
        def get_splits(self, n_test=0.25):
            # determine sizes
            test_size = round(n_test * len(self.X))
            train_size = len(self.X) - test_size
            # calculate the split
            return random_split(self, [train_size, test_size])

    # model definition
    class MLP(Module):
        # define model elements
        def __init__(self, n_inputs):
            super(MLP, self).__init__()
            # input to first hidden layer
            self.hidden1 = Linear(n_inputs, 140)
            kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
            self.act1 = ReLU()
            # second hidden layer
            self.hidden2 = Linear(140, 20)
            kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
            self.act2 = ReLU()
            # third hidden layer and output
            self.hidden3 = Linear(20, 1)
            xavier_uniform_(self.hidden3.weight)
            self.act3 = Sigmoid()

        # forward propagate input
        def forward(self, X):
            # input to first hidden layer
            X = self.hidden1(X)
            X = self.act1(X)
             # second hidden layer
            X = self.hidden2(X)
            X = self.act2(X)
            # third hidden layer and output
            X = self.hidden3(X)
            X = self.act3(X)
            return X

    # prepare the dataset
    def prepare_data(path):
```

```
        # load the dataset
        dataset = CSVDataset(path)
        # calculate split
        train, test = dataset.get_splits()
        # prepare data loaders
        train_dl = DataLoader(train, batch_size=32, shuffle=True)
        test_dl = DataLoader(test, batch_size=1024, shuffle=False)
        return train_dl, test_dl

    # train the model
    def train_model(train_dl, model):
        # define the optimization
        criterion = MSELoss()
        optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
        # enumerate epochs
        for epoch in range(100):
            # enumerate mini batches
            for i, (inputs, targets) in enumerate(train_dl):
                # clear the gradients
                optimizer.zero_grad()
                # compute the model output
                yhat = model(inputs)
                # calculate loss
                loss = criterion(yhat, targets)
                # credit assignment
                loss.backward()
                # update model weights
                optimizer.step()

    # evaluate the model
    def evaluate_model(test_dl, model):
        predictions, actuals = list(), list()
        for i, (inputs, targets) in enumerate(test_dl):
            # evaluate the model on the test set
            yhat = model(inputs)
            # retrieve numpy array
            yhat = yhat.detach().numpy()
            actual = targets.numpy()
            actual = actual.reshape((len(actual), 1))
            # round to class values
            yhat = yhat.round()
            # store
            predictions.append(yhat)
            actuals.append(actual)
        predictions, actuals = vstack(predictions), vstack(actuals)
        # calculate accuracy
        acc = accuracy_score(actuals, predictions)
        return acc

    # make a class prediction for one row of data
    def predict_model(test_dl, model):
```

```
        temp_df = pd.DataFrame()
        predictions, actuals = list(), list()
        for i, (inputs, targets) in enumerate(test_dl):
            # evaluate the model on the test set
            yhat = model(inputs)
            # retrieve numpy array
            yhat = yhat.detach().numpy()
            actual = targets.numpy()
            actual = actual.reshape((len(actual), 1))
            # round to class values
            yhat = yhat.round()
            # store
            predictions.append(yhat)
            actuals.append(actual)
        predictions = predictions[0].reshape(len(predictions[0])).tolist()
        actuals = actuals[0].reshape(len(actuals[0])).tolist()
        temp_df['pred'] = predictions
        temp_df['actual'] = actuals
        return temp_df

    # prepare the data
    path = '/content/drive/MyDrive/processed_training_data.csv'
    train_dl, test_dl = prepare_data(path)
    print(len(train_dl.dataset), len(test_dl.dataset))
    # define the network
    model = MLP(74)
    # train the model
    train_model(train_dl, model)
    # getting test results
    output_df = predict_model(test_dl, model)
    # evaluate the model
    acc = evaluate_model(test_dl, model)
    print('Accuracy: %.3f' % acc)
```

✓ 3s completed at 10:42 PM