Packages

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import recall_score, make_scorer, confusion_matrix,
roc_curve, auc
from sklearn.utils.multiclass import unique_labels
from sklearn.preprocessing import StandardScaler
import torch
```

Supervised Learning, Linear Model and Loss Function

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression

- Read in the data `>df = pd.read_csv(filename)`
- Scatter the data `>df.plot.scatter('x','y', ax = ax, alpha = 0.5)`
- Predict Model `>ypred = model.predict(X)`
- Loss function (tells how bad a fit is)
  1.Get model prediction `>yp = np.dot(X,b)`
  2.Get the vector of residuals `>res = y-predY`
  3.Get the processed residual OLS`>sum(res**2)` LAD`> sum(abs(res))`
  4.Get the gradient
  OLS`> -2*np.dot(res,X)`
  LAD`> sres = np.sign(res); grad =- (np.dot(sres,X))`
- Model fitting(selecting the parameters that minimizes the loss function)
  1.b has to have the same dimension as X has columns `>betas=np.zeros((ncols,1))`
  2.Optimize the loss `>RES =so.minimize(lossfcn,betas,args=(X,y),jac=True)`
  3.Obtain estimates from the optimizer `>estimated_betas=RES.x`

Maximum Likelihood

- Negative log likelihood: Construct the function as instructed.
- Regression Negative log likelihood: Process necessary parameters and call Negative log likelihood inside the function.
  `>return -1*betaloglik(y, mu, phi)`
- Maximum likelihood estimates:
  if an intercept is required, add a column of ones before the matrix X
  `>X = np.c_[np.ones((n,1)),df.drop('prate', axis='columns').values]`
  1. Initialize parameters required by the function
     ```
     > beta_start = np.zeros(X.shape[1])
     > phistart = np.ones((1,1))
     ```
  2. Call the optimizer
     ```
     >optimization = minimize(betaregnegloglik, x0=thetastart,
                              args = (X,y), method = 'Nelder-Mead')
     ```
  3. Retrieve the parameters from optimizer result
     `>results = optimization.x`

Classification and Evaluation

- Dataset Manipulation Techniques

Linspace: > `numpy.linspace(start, stop, num=50)`
Columns: > `df.shape[1]` Rows: > `df.shape[0]` Initialize y: > `y = df.Class.values`
Drop One column: > `X = df.drop('Class', axis='columns').values`
Select a column: > `gestation=df['gestation']` Initialize empty list: > `pre_term = []`
TrainTest Spilt: `>Xtrain,Xtest,ytrain,ytest = train_test_split(X,y,test_size=0.5, random_state=0)` Append(add) to list:> `pre_term.append(0)`
Get Dummies: `>model_data['work_rate_att'] = pd.Categorical(model_data.work_rate_att, categories=['Low','Medium','High'])`

- Sigmoid: Inputs: $(-\infty, \infty)$ Outputs: (0,1)
  `>eta = np.dot(X,b)`
  `>predictions = 1.0 / (1+np.exp(-eta))`
- Evaluation

1. Identify model and fill in parameters `>LogisticRegression(penalty='none')`
2. Identify design matrix
   > `X = df.drop(['preterm','gestation','bwt'], axis='columns').values`
3. Identify output y >`y = df.preterm.values`
4. Fit the model >`model.fit(X,y)`
5. We can retrieve the model coefficients
   >`print(model.coef_) print(model.intercept_)`
6. We can also predict the y data with fitted model `>yp1=lr_amount.predict(xp)`
7. We can also get the predict probability `>yp2=lr_amount.predict_proba(xp)`
8. For AUROC or other metrics `> roc_auc_score(y, ypred_prob)`
   `>accuracy_score(y, ypred)`
   https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

- To plot ROC curve

1.> `fpr, tpr, _ = roc_curve(ytest, ytest_prob_amount[:,1], pos_label=1)`
2.>`fig,ax=plt.subplots()` 3.>`ax.plot(fpr,tpr)`

Confidence Interval & Bootstrap

- Bootstrap Coefficients

1. define the measure type
   `regr = sklearn.linear_model.LinearRegression()`
2. define the result storage structure i.e. what you wanna return
   ```
   numboot = 1000
   n = len(data)
   theta = np.zeros((numboot,3))
   ```
3. >`for i in range (numboot)` resamples the data
   `>d = data.sample(n, replace=True)`
4. fit the model with resampled data
   ```
   >X_fit = np.c_[d.wt,d.wt**2]
   >regr.fit(X_fit,d.overall)
   ```
5. After fitted the model, extract the desired values, i.e. coefficients, predicted y based on other design matrix, predicted probability based on other design matrix etc.
   ```
   >y_boot_prob = LogPipeline.predict_proba(test.iloc[:, :-1])
   >fpr, tpr, thresholds = roc_curve(test.iloc[:, -1], y_boot_prob[:, 1], pos_label=1)
   >[i,0]=regr.intercept_theta>[i,1]=regr.coef_[0]>theta[i,2]=regr.coef_[1=
   ```
6. Plot the graph
   `sns.histplot(auc_out)`

- Confidence Interval

```
auc_dev = np.std(auc_out)
auc_min = log_test_auc - 1.96 * auc_dev
auc_max = log_test_auc + 1.96 * auc_dev
```

Model Selection & Cross Validation

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler.fit_transform
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

- Cross Validation

```
cross_val_score(model, X, y, cv = 10, scoring = 'mean_squared_error'))
```

`model`: estimator object implementing 'fit' `cv` = cross-validation splitting strategy
`X`=The data to fit. `y`=The target variable to try to predict
`scoring` =a scorer callable object / function with signature scorer(estimator, X, y)

- Pipeline

```
pipe = Pipeline([('preprocess', preprocess),('reg', LinearRegression())])
```

- Generalization Error and Confidence interval

```
>test_errors = np.power(ytest - ypred, 2)
>generalization_error = test_errors.mean()
>test_ci = generalization_error + 1.96 * np.std(test_errors) /
np.sqrt(len(test_errors)) * np.array([-1, 1])
```

Feature Selection & Regularzation

- Standardization

Make Scaler object > `StdScl = StandardScaler()`
Scale the design matrix > `std_Xtrain = StdScl.fit_transform(Xtrain)`

- Expand Design Matrix in a polynomial manner
  https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

```
>poly = sk.preprocessing.PolynomialFeatures(2, include_bias=False)
>X_new_train = poly.fit_transform(Xtrain)
```

- Coefficient Tuning with GridSearchCV()
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

1. Identify param_grid as per instruction

```
>alpha = np.linspace(0.1,1,20)
>param_grid = {'las__alpha': alpha}
```

2. Define estimator

```
>lassomodel = Pipeline
([('preprocess', preprocess),('las', Lasso(fit_intercept=True))])
```

3. Perform GridSearchCV()

```
>gscv = GridSearchCV(lassomodel, param_grid=param_grid, cv = 10,
scoring = 'neg_mean_squared_error')
```

4. Fit the model

```
>gscv.fit(X,y);
```

5. Extract and render result

```
>cv_results = pd.DataFrame(gscv.cv_results_)
>lam = cv_results.param_las__alpha.values
>rmse = np.sqrt(-1*cv_results.mean_test_score)
```

To See the available parameters for GridSearchCV

```
>print(lasso_pipe.get_params().keys())
>param_grid = {'logreg__C': 1/lam}
```

Enumerate Usage

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for counter, value in enumerate(my_list):
    print counter, value
# Output:# 0 apple# 1 banana# 2 grapes# 3 pear
```

Basic Concepts
Bias: Systematic Difference of the best fitted model from the true relationship
As the model becomes complex enough to model, the bias disappears
Variance: the fit around the average fit
Overfitting: While training error decreases, test error increases
Precision = #True Positive/#Predicted Positive
What proportion of the instances I labeled positive are actually positive?
Recall = #True Positive/#Class Positive(True Positive Rate)
What proportion of the positives in the population do I correctly identify?
F-measure = 2(Precision*Recall/ Precision+Recall)
Sensitivity = #True positive / #Class positive (True Positive Rate)
What proportion of the positives in the population do I correctly label?
Specificity = #True negative / #Class negative(True Negative Rate)
What proportion of the negatives in the population do I correctly label?

Convert np.array to pd.dataframe(After z-transform)

```
normalizer = StandardScaler()
RFM_data =  pd.DataFrame(normalizer.fit_transform(RFM_data),
index=RFM_data.index, columns=RFM_data.columns)
```

Post-Midterm

Tree Method

- Random Forest
  1. Define random forest

```
>energy_forest =
RandomForestRegressor(n_estimators=100,criterion='mse',max_depth=Non
e,min_samples_split=2,min_samples_leaf=1,min_weight_fraction_leaf=0.
0,max_features='sqrt',max_leaf_nodes=None,min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=True,
random_state=20210302,verbose=0,warm_start=True)
```

  2. Fit the model, and Calculate error over test set

```
> energy_forest.fit(x_train, y_train)
> pred_test = energy_forest.predict(x_test)
> mean_absolute_percentage_error(y_test, pred_test)*100)
```

- XGBoosting model

1) Define XGB model

```
>energy_XGB =XGBRegressor(max_depth=  learning_rate= n_estimators=)
```

2) Fit

```
>energy_XGB.fit(x_train, y_train)
```

3) Obtain Variable Importance

```
>importances = energy_XGB.feature_importances_
>indices = np.argsort(importances)[::-1]
>f, ax = plt.subplots(figsize=(3, 8))
>plt.title("Variable Importance - XGB")
>sns.set_color_codes("pastel")
>sns.barplot(y=[x_train.columns[i] for i in indices],
x=importances[indices],
  label="Total", color="b")
>sns.despine(left=True, bottom=True)
```

4) Calculate error over test set

```
>pred_test = energy_XGB.predict(x_test)
```

5) Create Plot

```
> pred_data = (pred_test, pred_extrapolation)
>test_data = (y_test, energy_extrapolation['Appliances'])
>colors = ("green", "red")
>groups = ("interpolation", "extrapolation")
>fig = plt.figure()
>ax = fig.add_subplot(1, 1, 1)
>for x, y, color, group in zip(test_data, pred_data, colors, groups):
  >ax.scatter(x, y, alpha=0.8, c=color, edgecolors='none', s=30,
label=group)
```

Neural Networks, Gradients, and Deep Learning

- Neural Networks Model Architecture

  Non-Linear Model

```
class NonLinearModel(torch.nn.Module):
    def __init__(self, input_size, hidden_size = 1024, num_classes=1):
        super().__init__()
        # Neural Network Architecture
        self.dense1 = torch.nn.Linear(in_features=num_features,
out_features=hidden_size)
        self.activation1 = torch.nn.ReLU()
        self.dense2 = torch.nn.Linear(in_features=hidden_size,
out_features=hidden_size)
        self.activation2 = torch.nn.ReLU()
        self.dense3 = torch.nn.Linear(in_features=hidden_size,
out_features=num_classes)
        self.activation3 = torch.nn.ReLU()
    def forward(self, X):
        X = self.dense1(X)
        X = self.activation1(X)
        X = self.dense2(X)
        X = self.activation2(X)
        X = self.dense3(X)
        X = self.activation3(X)
        return X
```

  Linear Model

```
class LinearModel(torch.nn.Module):

    def __init__(self, input_size, num_classes):

        super().__init__()

        # Neural Network Architecture
        self.dense1 = torch.nn.Linear(in_features=num_features,
out_features=num_classes)
        self.activation = torch.nn.LogSigmoid()

    def forward(self, X):
        X = self.dense1(X)
        X = self.activation(X)
        return X
```

  Define Model

```
model2 = NonLinearModel(num_features, num_classes)
```

- Train the model

Convert dataset into tensors

```
Xt = torch.FloatTensor(X)
```

```
yt = torch.LongTensor(y)
y_pred = model.forward(Xt)
```

1. Define maximum iterations, optimizer target and optimizing strategy
```
>max_iter = 100
>optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
>criterion = torch.nn.MSELoss()
```
2. Create lists for errors
```
>loss_list = []>mse_list = []>valid_list = []
```
3. In range of maximum iterations
```
>for i in range(max_iter):
```
4. Set the initial loss to 0
```
>epoch_loss = 0
```
5. Obtain the data sample from Dataloader
```
>for index, (data, label) in enumerate(trainloader):
```
6. Set the gradients to zero before starting to do back propragation(loss.backward())
```
>optimizer.zero_grad()
```
7. Obtain predicted value from model
```
>y_pred = model(data.cuda())
```
8. Calculate the loss
```
>loss = criterion(input=y_pred, target=label.reshape(-1, 1))
```
9. Call backward function, accumulates the gradient (by addition) for each parameter.
```
>loss.backward()
```
10. Step the optimizer, parameter update based on the current gradient and the update rule
```
>optimizer.step()
```
11. Update loss
```
>epoch_loss += loss.item()
```
12. Append the loss into the error list under no-gradient condition

```
>with torch.no_grad():
    >loss_list.append(epoch_loss)
    >y_pred = model.forward(torch.Tensor(train[:, :-1]).cuda())
    >mse_list.append(mean_squared_error(y_true=train[:, -1],
    y_pred=y_pred.cpu()))
    >y_pred_val = model.forward(torch.Tensor(test[:, :-1]).cuda())
    >valid_list.append(mean_squared_error(y_true=test[:, -1],
    y_pred=y_pred_val.cpu()))
```

13. Create live plot, or plot it at final stage
```
>live_plot(np.array(loss_list), np.array(mse_list), valid_list)
> plt.plot(np.arange(max_iter),loss_list)
```

Taking an optimization step

```
>for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Dimensionality Reduction

Variance – A measure of the spread of the data in a dataset with mean X

Covariance – a measure of how much each of the dimensions varies from the mean with respect to each other.

- PCA

  Initialization

```
>n_components == min(n_samples, n_features)
>nPCA = PCA(n_components = 200)
>nPCA.fit(TfIDF_train)
>variance = nPCA.explained_variance_ratio_
>variance =
np.cumsum(np.round(nPCA.explained_variance_ratio_,decimals=3)*100)
```

  Transformation

```
>Q1_X_test_tranformed = nPCA.transform(TfIDF_test)
```

- Autoencoder

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(236,100),
            nn.ReLU(),
            nn.Linear(100, 60),
            nn.ReLU(),
            )
        self.decoder = nn.Sequential(
            nn.Linear(60,100),
            nn.ReLU(),
            nn.Linear(100, 236),
            nn.ReLU())
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
return x
```

1) Build Model

```
auto_enc_11 = autoencoder()
```

2) Define Loss Function

```
loss_fn = torch.nn.MSELoss()
```

3) Define Learning Rate

```
learning_rate = 1e-1
```

4) Convert Dataset To Tensors

```
x_train_tensor = torch.from_numpy(TfIDF_train).float()
x_test_tensor = torch.from_numpy(TfIDF_test).float()
```

5) Create error lists

```
loss_per_batch_train = []
loss_per_batch_test = []
```

6) Within the range, encode the training set first

```
for t in range(5000):
    train_aut = auto_enc_11(x_train_tensor)
    loss = loss_fn(x_train_tensor, train_aut)
```

7) Every 100 times plot both train and test

```
if t % 100 == 99:
        test_aut = auto_enc_11(x_test_tensor)
        loss_test = loss_fn(x_test_tensor, test_aut)
        loss_per_batch_train.append(loss.item())
        loss_per_batch_test.append(loss_test)
```

8) Calibrate the autoencoder and accumulate each parameter

```
    auto_enc_11.zero_grad()
```

```
loss.backward()
```

9) Update parameter based on learning rate under no-gradient condition

```
with torch.no_grad():
      for param in auto_enc_11.parameters():
          param -= learning_rate * param.grad
```

10) Plot error

```
plt.plot(loss_per_batch_train, label='Training loss')
plt.plot(loss_per_batch_test, label='Test loss')
```

Clustering

K-Means Clustering

```
clusterer = KMeans(n_clusters=n_clusters, random_state=10)
cluster_labels = clusterer.fit_predict(RFM_data)
```

Silhouette Average

```
silhouette_avg = silhouette_score(RFM_data, cluster_labels)
```

Plot the clusters

```
for i in range(n_clusters):
  color = cm.nipy_spectral(float(i) / n_clusters)
  plt.scatter(PCA_data[cluster_labels==i, 0],PCA_data[cluster_labels==i, 1],
              label='Cluster %i' % (i+1))
```

Deploying Models

Covariate Shift is one term used to describe the situation where the test input distribution (in our case the distribution of X and D) is different from the training input distribution.

Confounding, missing information leads to different result. is the data are created experimentally by manipulating d (e.g. by randomizing d and watching what happens), then systematic bias from confounding can be eliminated.