Intro ML Review

Fraida Fund

Contents

Intro ML review

(Not comprehensive!)

What to know about each model

Make decisions about setting it up

Compute its output

Train for one step

What kinds of relationships it can learn

What it costs for training/inference

What insight we get from it, once trained

e.g. what its parameters mean

How to manage its bias/variance

What the "pieces" mean

What the visualizations mean

Not models, but know about...

Metrics

Evaluation and cross validation

Gradient descent

Feature selection/weighting

Hyperparameter optimization

Case studies to review

Data leakage case studies

- Beauty in the Classroom
- COVID Case Prediction
- Pre-term Birth Prediction
- Husky vs. Wolf Classification

Fairness and bias case studies

- COMPAS Recidivism Prediction
- Gender Bias in Word Embeddings

Using learned coefficients case studies

- Beauty in the Classroom
- Advertising + PCA Follow-up
- Husky vs. Wolf Classification

Re-thinking “good enough” case studies

- Beauty in the Classroom
- ICU Mortality Prediction

Moving beyond “out of the box” case studies

- Reading a Monkey Brain
- Voter Classification
- UAV-assisted Wireless Localization

Code

Getting data

`pandas`: read data from file

```
df = pd.read_csv("data.csv")
```

```
df = pd.read_table("data.txt", sep="\t")
```

```
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
```

Common arguments: `sep`, `header`, `index_col`

`numpy`: read data from file

```
X = np.load("X.npy")
```

`pandas` to `numpy`

```
X = df.to_numpy()
```

```
X = df.values # older way
```

`numpy` to `pandas`

```
df = pd.DataFrame(X, columns=["a","b","c"])
```

`pandas`: stacking columns

```
df = pd.concat([df1, df2], axis=1)
```

(Seen in: L6)

`numpy`: stacking columns

```
X = np.column_stack([X1, X2])
```

`numpy`: reshape 2D to 1D

```
x = X.reshape(-1,)       # flatten to 1D
```

`numpy`: reshape 1D to 2D

```
X = x.reshape(-1, 1)   # shape (n, 1)
```

```
X = x.reshape(1, -1)   # shape (1, n)
```

`numpy`: statistics by axis

```
m = np.mean(X, axis=0)     # mean of each column
```

```
m = np.mean(X, axis=1)     # mean of each row
```

Common statistics: `std`, `min`, `max`, `quantile`...

Preprocessing data

Ordinal encoding

```
map_dict = {'18-29': 1, '30-44': 2, '45-64': 3, '65+': 4}
df_enc_ord = pd.DataFrame(
    {'age': df['age'].map( map_dict) },
    index = df.index
)
```

(Seen in: L6)

One-hot encoding

```
# encode one column
df2 = pd.get_dummies(df["col"], dtype=np.int32)
```

(Seen in: L6)

`pandas`: converting string to datetime

```
df["date"] = pd.to_datetime(df["date"])
```

```
df["date"] = pd.to_datetime(df["date"], format="%Y-%m-%d")
```

(Seen in: Week 1 Exploratory Data Analysis Colab lesson + a few times in HW)

`pandas`: sort by value of column

```
df2 = df.sort_values(by="col")
```

```
df2 = df.sort_values(by=["col1", "col2"])
```

Common arguments: `ascending`

(Seen in: Week 1 Exploratory Data Analysis Colab lesson, Week 4 Model Selection Colab lesson, H4 K-fold CV with Fourier basis expansion)

`numpy`: sort by value of column

```
idx = np.argsort(X[:, 0])
X2 = X[idx]
```

```
# sort by col0, then col1
idx = np.lexsort((X[:, 1], X[:, 0]))
X2 = X[idx]
```

(Seen in: L6 when breaking ties)

`pandas`: drop missing values

```
# drop rows with ANY missing values
df2 = df.dropna()
```

```
# drop rows missing specific column by name
df2 = df.dropna(subset=["col1"])
```

```
# drop rows missing in either column
df2 = df.dropna(subset=["col1","col2"])
```

(Seen in: L6 when computing feature weights, H4 TimeSeriesSplit question)

`numpy`: drop missing values

```
# drop rows with ANY missing values
mask = np.isnan(X).sum(axis=1) == 0
X2 = X[mask]
```

```
# drop rows missing in either column by index
cols = [0, 2]
mask = np.isnan(X[:, cols]).sum(axis=1) == 0
X2 = X[mask]
```

`pandas`: impute missing values

```
# forward fill, data must be sorted
df["col"] = df["col"].fillna(method="ffill")
```

```
# using a statistic, only use stats of training set
s = df_tr["col"].mean() # use training data only
df_tr["col"] = df_tr["col"].fillna(s)
df_ts["col"] = df_ts["col"].fillna(s)
```

```
# using a constant
df["col"] = df["col"].fillna(0)
```

(Seen in: Week 1 Exploratory Data Analysis Colab lesson, H5 ICU mortality prediction)

`numpy`: **impute missing values**

```
# applies to all columns
# using a statistic, only use stats of training set
s = np.nanmean(Xtr, axis=0) # use training data only
Xtr2 = np.where(np.isnan(Xtr), s, Xtr)
Xts2 = np.where(np.isnan(Xts), s, Xts)
```

```
# applies to all columns
# using a constant
X2 = np.where(np.isnan(X), 0, X)
```

`numpy`: **Standardize**

```
x_mean = np.mean(X[idx_tr], axis=0)
x_std  = np.std(X[idx_tr], axis=0)
Xtr_std = (X[idx_tr] - x_mean) / x_std
Xts_std = (X[idx_ts] - x_mean) / x_std
```

`scikit-learn`: **Standardize**

```
scaler = StandardScaler()
Xtr_std = scaler.fit_transform(X[idx_tr])
Xts_std = scaler.transform(X[idx_ts])
```

(Seen in: Week 4 Regularization Colab lesson)

`numpy`: **Min-max scale**

```
x_min = np.min(X[idx_tr], axis=0)
x_max = np.max(X[idx_tr], axis=0)
Xtr_mm = (X[idx_tr] - x_min) / (x_max - x_min)
Xts_mm = (X[idx_ts] - x_min) / (x_max - x_min)
```

`scikit-learn`: **Min-max scale**

```
scaler = MinMaxScaler()
Xtr_mm = scaler.fit_transform(X[idx_tr])
Xts_mm = scaler.transform(X[idx_ts])
```

(Seen in: L6)

`scikit-learn`: `CountVectorizer`

```
vect = CountVectorizer()
Xtr = vect.fit_transform(text_tr)      # fit on training text
Xts = vect.transform(text_ts)          # transform test text
```

Common arguments: `stop_words`

(Seen in: H5)

Create "transformed" features

```
df = df.assign(interaction = df["col1"] * df["col2"])
```

```
# interaction of col0 and col1
interaction = X[:, 0] * X[:, 1]
X_new = np.column_stack([X, interaction])
```

(Seen in: L2)

Oversampling/undersampling

```
ovr = ADASYN(n_neighbors = 5, random_state = random_state)
X_ovr, y_ovr = ovr.fit_resample(X, y)
```

(Seen in: H8)

Slicing and selecting

`pandas`: select columns

```
col = df["col"]                 # one column (Series)
cols = df[["col1", "col2"]]     # list of columns (DataFrame)
```

`numpy`: select columns

```
x = X[:, 3]          # one column by index
x = X[:, 2:5]        # contiguous columns (2,3,4)
x = X[:, [0, 3, 7]]  # list of columns
```

`pandas`: select rows

```
# by integer position - use .iloc
rows = df.iloc[5]
```

```
# by condition
rows = df[df["cola"] > 0]
rows = df[(df["cola"] > 0) & (df["colb"] < 5)]
```

`numpy`: select rows

```
r = X[5]          # one row
r = X[10:20]      # slice of rows
r = X[[0, 3, 7]]  # list of row indices
```

```
# by condition on column 0
mask = X[:, 0] > 0
X2 = X[mask]
```

```python
# by condition using argwhere
idx = np.argwhere(X[:, 0] > 0).reshape(-1)
X2 = X[idx]
```

`sklearn`: **train_test_split**

```python
# split X and y together
Xtr, Xts, ytr, yts = train_test_split(
    X, y, test_size=0.2, random_state=0, shuffle=True
)
```

```python
# split X only
Xtr, Xts = train_test_split(X, test_size=100, shuffle=False)
```

Common arguments: `test_size` (ratio or number), `shuffle`, `random_state`

`sklearn`: **GroupShuffleSplit**

```python
gss = GroupShuffleSplit(test_size=0.2, random_state=0)
train_idx, test_idx = next(gss.split(X, y, groups))
```

`sklearn`: **StratifiedShuffleSplit**

```python
sss = StratifiedShuffleSplit(test_size=0.2, random_state=0)
train_idx, test_idx = next(sss.split(X, y))
```

`python`: **loop with index and value**

```python
for i, x in enumerate(X):
    # do stuff with i = index, x = row (X is 2D)
```

`sklearn`: **KFold with `numpy` array**

```python
kf = KFold(n_splits=5, shuffle=True, random_state=0)

for i, (idx_tr, idx_ts) in enumerate(kf.split(X)):
    Xtr, Xts = X[idx_tr], X[idx_ts]
```

`sklearn`: **KFold with `pandas` data frame**

```python
kf = KFold(n_splits=5, shuffle=True, random_state=0)

for i, (idx_tr, idx_ts) in enumerate(kf.split(X)):
    Xtr, Xts = X.iloc[idx_tr], X.iloc[idx_ts]
```

Note the use of `iloc`! (Applies to all the “variants,” too.)

`sklearn`: TimeSeriesSplit

```
tscv = TimeSeriesSplit(n_splits=5)

for i, (idx_tr, idx_ts) in enumerate(tscv.split(X)):
    Xtr, Xts = X[idx_tr], X[idx_ts]
```

`sklearn`: GroupKFold

```
gkf = GroupKFold(n_splits=5)

for i, (idx_tr, idx_ts) in enumerate(gkf.split(X, groups)):
    Xtr, Xts = X[idx_tr], X[idx_ts]
```

Comment: groups by `groups`

`sklearn`: StratifiedKFold

```
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)

for i, (idx_tr, idx_ts) in enumerate(skf.split(X, y)):
    Xtr, Xts = X[idx_tr], X[idx_ts]
```

Comment: stratifies by `y`

`sklearn`: GridSearchCV

```
param_grid = [
  {'C': [1e-1, 1e1, 1e3], 'gamma': [1e2, 1, 1e-1, 1e-3]},
 ]
grid = GridSearchCV(model, param_grid, cv=5)
grid.fit(Xtr, ytr)
best = grid.best_params_
```

(Seen in: H8)

One-SE rule

- **Step 1**: Find models within one SE of “best” model
- **Step 2**: Pick the simplest model from the list of candidates

One-SE rule: Step 1 with lower-is-better metric

```
# score_val is a lower-is-better metric
# so threshold is: SMALLEST mean score PLUS its std/(nfold-1)
idx_min = np.argmin(score_val.mean(axis=1))
target = score_val[idx_min,:].mean() + score_val[idx_min,:].std()/np.sqrt(nfold-1)
# candidate models have mean score BELOW that threshold
idx_one_se = np.where(score_val.mean(axis=1) <= target)
```

One-SE rule: Step 1 with higher-is-better metric

```
# score_val is a higher-is-better metric
# so threshold is: LARGEST mean score MINUS its std/(nfold-1)
idx_min = np.argmax(score_val.mean(axis=1))
target = score_val[idx_min,:].mean() - score_val[idx_min,:].std()/np.sqrt(nfold-1)
# candidate models have mean score ABOVE that threshold
idx_one_se = np.where(score_val.mean(axis=1) >= target)
```

One-SE rule: Step 2 with models ordered from least complex to most

```
# get LOWEST indexed model (least complex)
m_one_se = np.min(model_list[idx_one_se])
```

One-SE rule: Step 2 with models ordered from most complex to least

```
# get HIGHEST indexed model (least complex)
m_one_se = np.max(model_list[idx_one_se])
```

Metrics

Regression metrics

```
mse = mean_squared_error(y, y_pred)
```

```
mae = mean_absolute_error(y, y_pred)
```

```
# r2_score(y_true, y_pred) - order matters: true first, pred second
r2  = r2_score(y, y_pred)
```

Comment: `y` and `y_pred` must have same shape...

`numpy`: prediction by mean

```
# ones * mean, works by broadcasting
m = np.mean(ytr)
pred_mean = np.ones(y.shape) * m
```

```
# fill array with mean
m = np.mean(ytr)
pred_mean = np.full_like(y, m)
```

```
# repeat mean
m = np.mean(ytr)
pred_mean = np.repeat(m, len(y))
```

Comment: use training data only to compute mean! then can fill all of `y` or just `ytr` or `yts`.

Classification metrics (label-based)

```python
acc   = accuracy_score(y, y_pred)
bacc  = balanced_accuracy_score(y, y_pred)
f1    = f1_score(y, y_pred)
prec  = precision_score(y, y_pred)
rec   = recall_score(y, y_pred)
```

Comment: `y` and `y_pred` must have same shape...

Classification metrics (probability-based)

```python
# y_proba = probability for class 1
y_proba = model.predict_proba(X)[:, 1]
auc  = roc_auc_score(y, y_proba)
```

`numpy`: prediction by mode

Same idea as "prediction by mean", but to get mode...

```python
vals, counts = np.unique(ytr, return_counts=True)
# index of most common value in counts array
m_idx = np.argmax(counts)
# actual label
m = vals[m_idx]
```

```python
from scipy.stats import mode
m = mode(ytr, keepdims=True).mode[0]
```

Comment: use training data only to compute mode!

(Seen in: H5 ICU Mortality Prediction, and others)

Model fitting and prediction

General pattern

```
# choose model + pass arguments
model = Model()
# train, Xtr MUST be 2D (n_samples, n_features)
model.fit(Xtr, ytr)
# predict
y_pred = model.predict(Xts)
```

(Seen in: Week 2 Colab lesson)

Linear models

```
model = LinearRegression()
model = Ridge(alpha=1.0)
model = Lasso(alpha=0.1)
model = LogisticRegression()
```

Nearest neighbors

```
model = KNeighborsRegressor(n_neighbors=5)
model = KNeighborsClassifier(n_neighbors=5)
```

Trees and ensembles of trees

```
model = DecisionTreeRegressor(max_depth=5)
model = DecisionTreeClassifier(max_depth=5)
```

```
model = RandomForestRegressor(n_estimators=200)
model = RandomForestClassifier(n_estimators=200)
```

```
model = AdaBoostClassifier(n_estimators=200, learning_rate=0.5)
```

SVM

```
model = SVC(kernel="rbf", C=1.0, gamma=0.1)
```

KMeans clustering

```
model = KMeans(n_clusters=3, random_state=0)
Xtr_cls = model.fit_predict(Xtr)         # unsupervised: no y
Xts_cls = model.predict(Xts) # cluster assignments on new data
```

PCA

```
pca = PCA(n_components=2)
Xtr_pca = pca.fit_transform(Xtr)     # fit PCA on training data
Xts_pca = pca.transform(Xts)       # apply to test data
```

Pytorch models

Model: Binary classification with prob output

```
nout = 1  # !!!

class SimpleNNBinaryClassification(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNBinaryClassification, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.output(x))  # !!!
        return x

model = SimpleNNBinaryClassification(nin, nh, nout)
loss_fn = nn.BCELoss() # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model: Binary classification with logit output

```
nout = 1  # !!!

class SimpleNNBinaryClassificationLogits(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNBinaryClassificationLogits, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = self.output(x) # !!!
        return x

model = SimpleNNBinaryClassificationLogits(nin, nh, nout)
loss_fn = nn.BCEWithLogitsLoss() # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model: Multi-class classification with log prob output

```
nout = 3 # !!!

class SimpleNNMultiClassification(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNMultiClassification, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.log_softmax(self.output(x), dim=1) # !!!
        return x

model = SimpleNNMultiClassification(nin, nh, nout)
loss_fn = nn.NLLLoss() # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model: Multi-class classification with logit output

```
nout = 3 # !!!

class SimpleNNMultiClassificationLogits(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNMultiClassificationLogits, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = self.output(x) # !!!
        return x

model = SimpleNNMultiClassificationLogits(nin, nh, nout)
loss_fn = nn.CrossEntropyLoss()  # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model: Regression with one output

```
nout = 1  # !!!

class SimpleNNRegression(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNRegression, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = self.output(x) # !!!
        return x

model = SimpleNNRegression(nin, nh, nout)
loss_fn = nn.MSELoss() # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Model: Regression with multi output

```
nout = 2  # !!!

class SimpleNNMultiRegression(nn.Module):

    def __init__(self, nin, nh, nout):
        super(SimpleNNMultiRegression, self).__init__()
        self.hidden = nn.Linear(nin, nh)
        self.output = nn.Linear(nh, nout)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = self.output(x) # !!!
        return x

model = SimpleNNMultiRegression(nin, nh, nout)
loss_fn = nn.MSELoss() # !!!
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Predictions: binary classification with prob output

```
with torch.no_grad():
    outputs = model(X)
    preds = (outputs > 0.5).int()
```

Predictions: binary classification with logit output

```
with torch.no_grad():
    logits = model(X)
    outputs = torch.sigmoid(logits)
    preds = (outputs > 0.5).int()
```

Predictions: multi-class classification, either output type

```
with torch.no_grad():
    outputs = model(X)
    preds = torch.argmax(outputs, dim=1)
```

Predictions: regression, any number of outputs

```
with torch.no_grad():
    outputs = model(X)
    preds = outputs
```

ConvNets: convolution blocks

```
# Before: (batch_size, n1, H1, W1)
nn.Conv2d(n1, n2, kernel_size=3, padding=0),
# After: (batch_size, n2, H2, W2)
nn.ReLU(),      # no change in shape
nn.BatchNorm2d(n2), # no change in shape
nn.MaxPool2d(kernel_size=2),
# After: (batch_size, n2, H3, W3)
```

ConvNets: classification head

```
self.classifier = nn.Sequential(
    # Before: (batch_size, n, h, w)
    nn.Flatten(),
    # After: (batch_size, n x h x w)
    nn.Linear(n_out_conv, n_out),
    # After: (batch_size, n_out)
)
```

General tips for autograding

- run your code beginning to end before submitting to grader
- don't re-use out-of-loop variable names inside loops
- keep an eye on `#grade` tags