1 Module 9: Model Selection & Regularization

Two approaches for model selection/hyperparameter tuning:

- 1. Sequential Feature Selection (Greedy)
- 2. Regularization (global optimization)

Contents

1.1 (Forward) Sequential Feature Selection

Algorithm:

- 1. For each feature, fit it individually.
- 2. Pick the best-performing feature and $\rightarrow \Phi_{\text{selected}}$ set of features
- 3. With the remaining features, fit them each but with a model including Φ_{selected} features
- 4. Continue adding features to Φ_{selected} until desired

This is a **"greedy" algorithm**: one which makes a series of greedy choice which are best at a particular moment – not guaranteed to yield the best solution, but is fast. As such, the result is liable to vary depending on the randomness of the training indices provided.

```
# First, create two arrays which encapsulate the indices of the dataset
indices = shuffle( range(0,len(data)) )
training_indices, dev_indices = np.split(indices, [int])
```

```
# Note that 'neg_mean_squared_error' is selected for scoring, as the selector
↪  intends to MAXIMIZE its value
feature_select = SequentialFeatureSelector(estimator = LinearRegression(),
↪  scoring = 'neg_mean_squared_error', cv = [[training_indices, dev_indices]],
↪  n_features_to_select = int)

# Display top n features selected
best_features = pd.DataFrame( feature_select.fit_transform(X, y), columns =
↪  feature_select.get_feature_names_out() )

# MAKE SURE when testing MSE to filter the data by the correct indices, e.g.
data = LinearRegression.fit( best_features.iloc[training_indices],
↪  y.iloc[training_indices] ).predict( best_features.iloc[dev_indices])

mean_squared_error(data, y.iloc[dev_indices])
```

Note REVERSE feature selection: originally fitting with ALL features and removing the worst-performing feature at a time.

1.2 L2 (Ridge) Regularization

An alternate approach for controlling complexity using a complexity parameter: α. The goal of regularization is to prevent overfitting.

Ridge or L2 Regularization: at a high level, the algorithm minimizes MSE as usual but penalizes the size of the coefficients by α:

$$\text{MSE} = \alpha(\theta_1^2 + \theta_2^2 + ...) + \frac{1}{n}\sum\left[y_i - \left(\sum_{j=1}^{d}\theta_j\phi_{i,j} + b\right)\right] \tag{1}$$

NOTE that this penalization will disproportionately affect features with LOWER values. Data must be Z-scored.

```
# Ridge is also a LinearRegression model
# Complexity parameter £\alpha \propto \frac{1}{\theta_i}£ for £\theta_i£
↪  corresponding to the coefficient of each feature. Essentially, increasing
↪  alpha decreases model complexity.
lm_model = Ridge(alpha = 100)
lm_model.fit(X, y)
```

1.3 Scaling data

Z-score: for each column, subtract the data by its mean and divide by its standard deviation

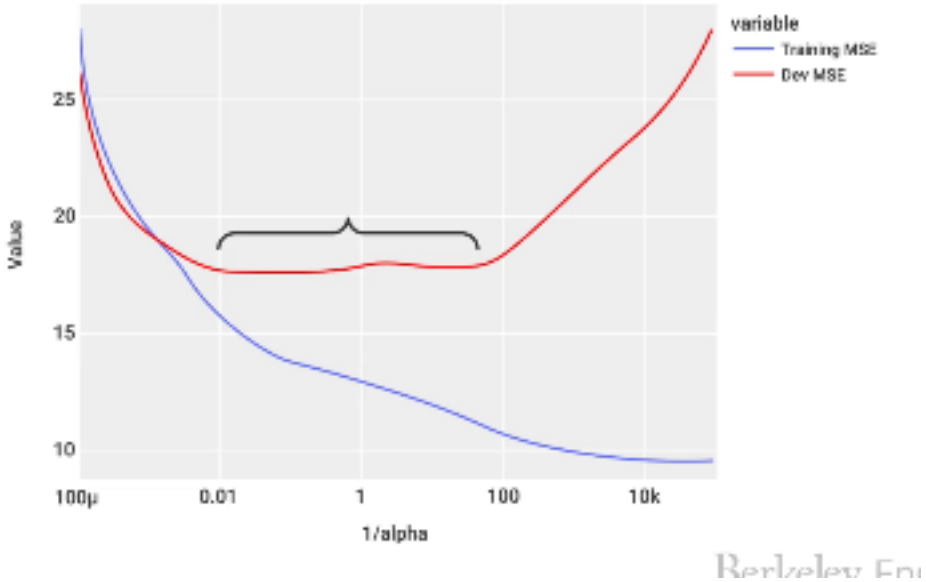
```
ss = StandardScaler()

# Rescale data
rescaled_data = pd.DataFrame( ss.fit_transform(data), columns =
↪  ss.get_feature_names_out() )
```

1.4 GridSearchCV

Allows for the tuning of hyperparameters automatically!

For example, we can vary α and watch the train/test scores change. There tend to be a swath of α values where the test score remains roughly the same.




```
# We start with a standard fitting process such as
model = Pipeline( ('poly', PolynomialFeatures()), ('scaler', StandardScaler()),
↪  ('reg', Ridge()) )

# Generate a dictionary of parameters to vary and their respective range
parameters = { 'alpha': np.logspace(0,10,10) }
```

```
from sklearn.model_selection import GridSearchCV

# Use the training/dev indices as before.
gs = GridSearchCV( estimator = model, param_grid = parameters, scoring =
↪  'neg_mean_squared_error', cv = [[training_indices, dev_indices]] )

gs.fit( X, y )

# You can investigate the best_model to find optimal hyperparameters
best_model = gs.best_estimator_

# Generate stats on fitting results/process
gs.cv_results_

"""
NOTE that gs.predict() uses the most RECENT model; not the best one.
Make sure to pull the best model, then predict with that.
"""
```

1.5 LASSO (L1) Regularization Regression

There are many other methods for penalizing with α:

$$\frac{1}{n}\sum\left[y_i - \left(\sum_{j=1}^{d}\theta_j\phi_{i,j} + b\right)\right] + ...$$

- L1 (Lasso): $\alpha\sum_{j=1}^{d}|\theta_j|$
- L2 (Ridge): $\alpha\sum_{j=1}^{d}\theta_j^2$

```
sklearn.linear_model import Lasso
```

With L1, many coefficients will go to zero. A relatively small number of features will remain, therefore L1 can be used for feature selection.

1.6 K-fold Cross Validation

Algorithm:

- 1. Define a model and the hyperparameter to be tuned.
- 2. Split dataset into k sets of data.

- 3. For each set, reserve it as the temporary validation set and use the rest to train.
- 4. Average the error computed k times.

Leave-one-out uses k = number of data points, meaning that each data point becomes itself a validation point.

```
# Set the cv constructor to be the integer k
GridSearchCV( estimator = model, param_grid = parameters, scoring =
↪  'neg_mean_squared_error', cv = k )
```

1.7 Applications

Quality/variety of data hugely affects the results of a model.

The Bandit problem: balance exploitation (selecting groups with proven track records) with exploration (selecting underrepresented groups to learn about quality)