# Module 9: Model Selection and Regularization

## Quick Reference Guide

### Learning Outcomes:

1. Identify the effect of polynomial features on Multidimensional data.
2. Utilize sequential feature selection to explore model complexity.
3. Apply regularization to control model complexity.
4. Use regularization parameter “alpha” in ridge regression.
5. Employ StandardScalar in ridge regularization.
6. Scale data using standardization to avoid over-penalizing features in a regularized model.
7. Apply GridSearchCV to select the optimal value for alpha.
8. Select features using Lasso regression.
9. Evaluate the differences between leave-one-out cross-validation, k-fold cross-validation, and holdout cross-validation.

### Polynomial Features on Multidimensional Data

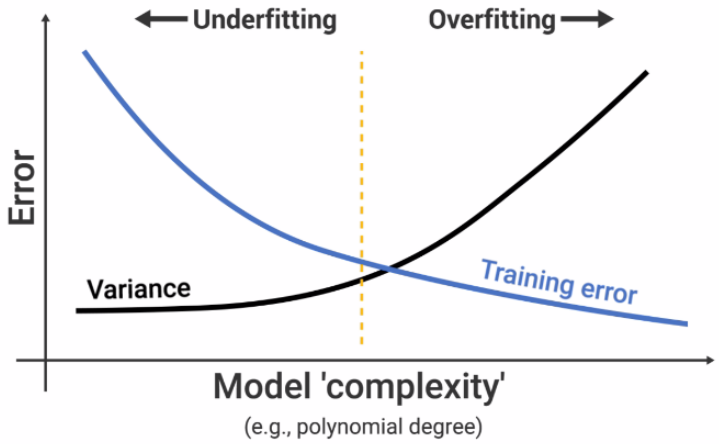
A **polynomial features** object can be applied to **multidimensional data**. The number of generated features depends on the degree of the polynomial features object and the number of features you started with.

For example, if the initial DataFrame has two columns, horsepower (hp) and weight, then the output of the polynomial features object with degree-two will have five features. The first two features are hp and weight. The remaining three features are $hp^2$, hp × weight, and $weight^2$. Note that all order-two combinations of the features are formed. If the polynomial features degree parameter is increased to three, it will yield nine output

features — the same five features as the degree-two model as well as $hp^3$, $hp^2 \times weight$, $hp \times weight^2$, and $weight^3$.

There are two approaches to conducting **simple cross-validation** for the above example. One approach is to find the mean squared error (MSE) for various choices of the hyperparameter degree. However, simply using cross-validation to select degree could fail to identify which parameter is useful for modeling.

A more exhaustive approach would be to consider every possible combination of parameters to see which yields the lowest MSE. If the dataset is not too large, it will not be computationally expensive. However, with a large dataset it could take years to exhaustively compute the MSE for all combinations. For instance, a dataset with five numerical features fed into a degree-three polynomial features object will yield a 55-dimensional feature set. To try out all possible features would require training $2^{55}$ models, which is a lot.




The space that includes the 55-dimensional model does not neatly fit into the conceptual framework represented by the idealized diagram of model error versus model complexity.

When only adjusting the polynomial degree, it was clear which model was more complex — just compare their degrees. Furthermore, there was only ever one model of any given degree. With $2^{55}$ possible models of varying complexity, the space which needs to be explored is much more complicated.

In this instance, you will focus on two alternate approaches that are faster than exhaustively searching all possible models — **sequential feature selection** and **regularization**.

## Sequential Feature Selection

With **sequential feature selection**, you can search a subset of the combination of the available features and add features iteratively.

To perform sequential feature selection:

- Start with zero features selected — $\Phi_{selected} = \{\}$
- Add features one at a time until the size of $\Phi_{selected}$ reaches some arbitrary predetermined number of features

To decide which feature to add at each step:

- Let $\Phi_{selected}$ be the set of features you have already decided to add
- Let $\Phi_{remaining}$ be the set of features that you have not yet added
- For each feature $\Phi_{candidate}$ in $\Phi_{remaining}$, fit a model that includes $\Phi_{selected}$ and $\Phi_{candidate}$
- Compute the development (validation) set MSE
- Whichever $\Phi_{candidate}$ has the lowest MSE is added to $\Phi_{selected}$

The process is repeated until the number of desired features is obtained.

For example, suppose you have five possible features — horsepower (hp), weight, $hp^2$, hp × weight, and $weight^2$ — and you want to select three.

Initially, no features are selected: $\Phi_{selected} = \{\}$.

And all features remain: $\Phi_{remaining} = \{hp, weight, hp^2, hp \times weight, weight^2\}$.

Then fit five models — one each for $hp$, $weight$, $hp^2$, $hp \times weight$, and $weight^2$ in this example — and compute the development (dev) set errors.

Suppose the $hp$ model has the lowest MSE. In that case, $hp$ is now your first feature and will stay selected for this procedure forever: $\Phi_{selected} = \{hp\}$. The $\Phi_{remaining} = \{weight, hp^2, hp \times weight, weight^2\}$.

Next, you fit four models and compute their dev set errors:

- A model using only $\{hp, weight\}$
- A model using only $\{hp, hp^2\}$
- A model using only $\{hp, hp \times weight\}$
- A model using only $\{hp, weight^2\}$

Suppose that $\{hp, weight\}$ has the lowest MSE. It is now moved into the selected feature set: $\Phi_{selected} = \{hp, weight\}$. And $\Phi_{remaining} = \{hp^2, hp \times weight, weight^2\}$.

To find the third parameter, you fit three models and compute their dev set errors. As before, each of these models has the two features from $\Phi_{selected}$, namely $hp$ and $weight$, and one additional feature from $\Phi_{remaining}$:

- A model using only $\{hp, weight, hp^2\}$

- A model using only $\{hp, weight, hp \times weight\}$
- A model using only $\{hp, weight, weight^2\}$

Suppose that of those three models, the one that contains $\{hp, weight, hp^2\}$ has the lowest MSE. The three features are now selected.

## Sequential Feature Selection in Scikit-Learn

The **sequential feature selection** transformer in scikit-learn takes a high-dimensional dataset and outputs a lower dimensional dataset using the sequential feature selection process.

To perform sequential feature selection in scikit-learn:

1. Generate all degree-three combinations of the numeric feature
2. Create two lists of numbers—one for all of the sample numbers for the training set and one for the sample numbers for the dev set
3. Create the sequential feature selection transformer – the sequential feature selection transformer constructor takes the following arguments:
   - **estimator** = (the untrained model to use)
   - **scoring**=(the loss function to use)
   - **cv**=(a list containing the training and dev sample indices)
   - **n_features_to_select** = (the number of features required)
4. Call fit_transform() on the data
5. Convert the output of fit_transform() into a DataFrame

The output features minimize the dev set error according to this sequential feature selection process.

Sequential feature selection algorithms are called **greedy algorithms** because:

- They make a series of greedy choices that are the best at a particular moment
- They are not guaranteed to find the global optimum, but they are generally fast

It is important to note:

- Sequential feature selection with n = 4, for instance, does not necessarily give the best choice of four features because only a fraction of the possible combinations is explored.
- Each time a feature is added, there is no guarantee that the dev set error will decrease. As features are added, the dev set error usually goes down at first, then fluctuates up and down as more features are added, before finally going back up when all the features are added, thus overfitting.
- Each time the sequential features selection process is run, different features may be selected. The process depends on the samples that are selected. Each time the process is run, different samples are selected; with different samples different features may be selected, especially if there is redundancy between some features.

Variants of sequential feature selection include:

- **Reverse feature selection**—start with all features and remove one at a time
- **Forward and reverse feature selection**—randomly add or remove a feature a fixed number of times

There is variance in the technique used to decide that one feature set is better than another.

## A First Look at Regularization

**Regularization** is an alternate approach for controlling the complexity of a model.

With regularization, you will:

- Keep all features, even if there are a huge number of them
- Adjust complexity using a single parameter: alpha ($\alpha$)

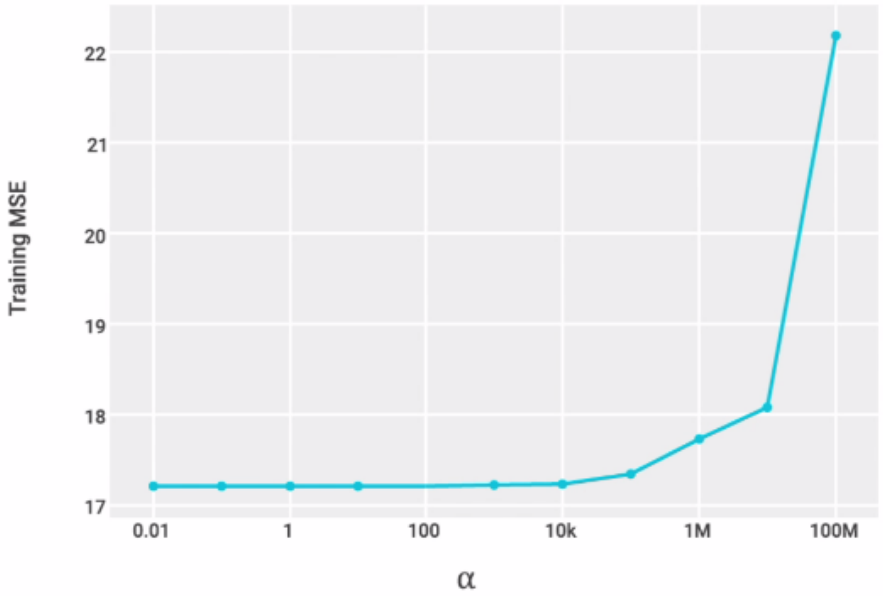
The basic syntax for a regularized model is:

*model name* = Ridge(alpha = *n*)
*model name*.fit()

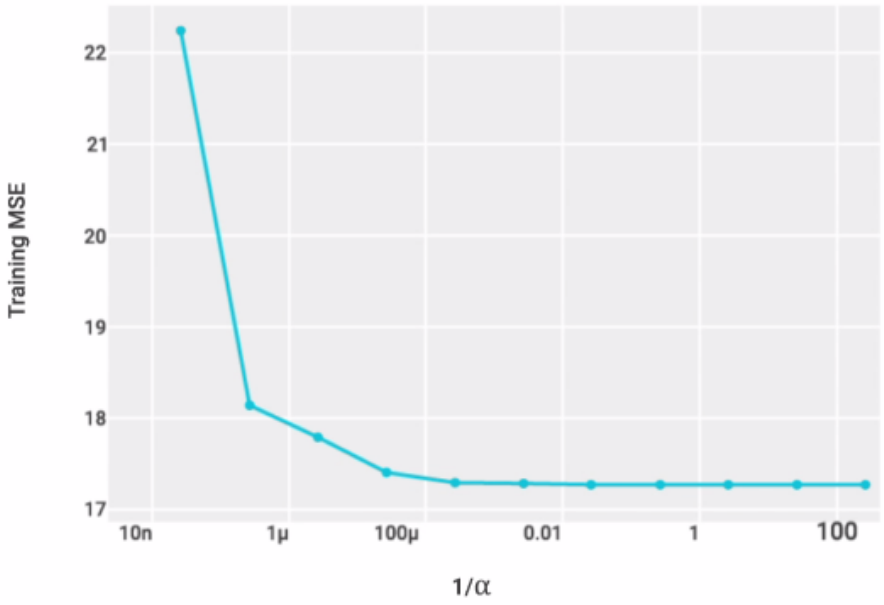
**Ridge** is a type of linear regression model. Ridge regression models are simpler. Even though the model still has the expressive power of all the input features, it loses its ability to use them as much.

You have to specify the $\alpha$ parameter, which tries to keep the complexity of the model in check. Different choices of $\alpha$ affect the parameters:

- As $\alpha$ increases, the model becomes less complex, and the $\theta$ parameters are constrained and shrink closer to zero. The training error increases as well.
- If $\alpha = 0$, the model is not constrained at all and it is just a standard linear regression.




This plot shows how the training error increases—and complexity decreases—as $\alpha$ increases.




If you plot $\frac{1}{\alpha}$, then the trend is reversed. There is a steeply dropping MSE as $\frac{1}{\alpha}$ increases. Then it levels off for a little bit, and then it becomes just a bit steeper before decaying to a fixed asymptote around MSE of 17.1. The

reason that the MSE levels off at that specific value is because it is the MSE of a standard linear regression model.

## How Regularization Works

**Ridge regression** tries to find $b$, $\theta_1$, $\theta_2$, until $\theta_d$, such that an equation is minimized. The model fitting procedure minimizes the sum of the mean squared error (MSE) plus alpha ($\alpha$) times the sum of the squares of the parameters. The new function that is minimized is the **objective function**.

The ridge regression equation is:

$$\alpha(\theta_1^2 + \theta_2^2 + \ldots + \theta_d^2) + \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(\sum_{j=1}^{d}\theta_j\phi_{i,j} + b\right)\right)^2$$

The **penalty term** is the $\alpha(\theta_1^2 + \theta_2^2 + \ldots + \theta_d^2)$ part of the equation. This is also the L2 norm of the model parameters – so the process of adding this term to a linear regression model is also sometimes called **L2 regularization**. The terms **ridge regression** and **L2 regularization** are interchangeable for most purposes.

The MSE is the $\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(\sum_{j=1}^{d}\theta_j\phi_{i,j} + b\right)\right)^2$ part of the equation.

This is the ridge regression equation in sigma notation:

$$\alpha\sum_{j=1}^{d}\theta_j^2 + \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(\sum_{j=1}^{d}\theta_j\phi_{i,j} + b\right)\right)^2$$

To use this equation in practice:

- Compute the predictions, $\hat{y}_1$ and $\hat{y}_2$, which is the $\left(\sum_{j=1}^{d} \theta_j \phi_{i,j} + b\right)$ portion of the equation
- Compute the MSE, which is the $\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(\sum_{j=1}^{d} \theta_j \phi_{i,j} + b\right)\right)^2$ portion of the equation
- Add the penalty term, which is the $(\theta_1^2 + \theta_2^2 + \ldots + \theta_d^2)$ portion of the equation

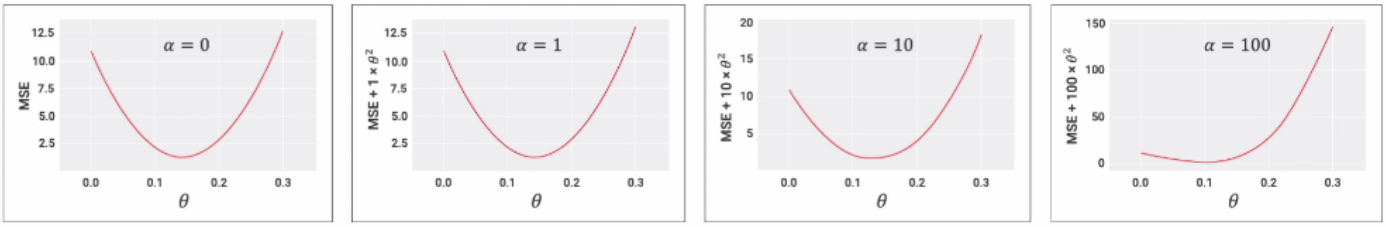
The answer is the objective function, the value that the model fitting procedure wants to minimize. Note that $\alpha$ is set in stone and is not allowed to change during fitting.

When you increase the size of $\alpha$, the parameters decrease. This process is called **shrinkage**. The penalty term establishes a tradeoff between the size of the parameters and the error of the model.

So, the $\alpha$ parameter sets a budget for the model:

- If $\alpha$ is very small, the budget is effectively unlimited
- If $\alpha$ is large, the penalty term becomes more sensitive to the magnitudes of the parameters and the model size budget becomes constrained

The impact of $\alpha$ on the MSE can be illustrated visually as follows:



As $\alpha$ increases, the curve becomes flatter, and the optimal parameter shifts. The objective function for high-dimensional models cannot be plotted, but something similar happens when you apply ridge regression in those spaces. As $\alpha$ grows, the location of the minimum of the objective function shifts closer and closer to the origin.

## Scaling

The **penalty term** affects all parameters equally, regardless of the scale of the parameters. If you have a limited parameter budget constrained by $\alpha$ ($\alpha \sum_{j=1}^{d} \theta_j^2$), it is cheaper to spend your budget on large features since the corresponding $\theta$ only needs to be relatively small to take advantage of it.

In other words, the penalty term more heavily penalizes features that are small in value.

**Standardization** is a technique to rescale data in the context of regularization. Said differently, it is a method to deal with the penalty term. To standardize a column of a DataFrame, you convert each feature into a Z-score (subtract out the mean of the column and then divide the result by the standard deviation of the column). This process is applied to each column of the DataFrame independently.

Scikit-learn provides a transformer that you can use to perform standardization called **StandardScaler()**.

To use the **StandardScaler()** transformer in scikit-learn:

- Instantiate a StandardScaler() object
- Call the fit_transform() method on the DataFrame

- Add columns = ss.get_feature_names_out() to turn the results into a DataFrame with the appropriate name

When all of the features are Z-scores, they are approximately the same scale. As a result, regularization penalties will be applied to each column more equitably.

When standardization is applied, the mean of each column is zero and the variance of each column is one. Also, no information is lost, so you can use the inverse transform method of the standard scaler to invert the transform.

A standard scaler can be placed in a pipeline to avoid the need to explicitly generate a DataFrame that has been rescaled. For example, you can augment the ridge model pipeline with a new scale step that comes after the polynomial feature step. So when you call fit() on the scale ridge model, the object will add all of the new features, then rescale by standardizing them, and pass the standardized data to the ridge regression model.

If you neglect to scale, you may get a warning. Neglecting to include a scaler in your pipeline may result in a model that is inferior.

## GridSearchCV

**GridSearchCV** is a scikit-learn estimator that lets you compare different hyperparameters, combinations of hyperparameters, and different models. It can compare alphas ($\alpha$), which are hyperparameters. The choice of $\alpha$ will affect the performance of the model.

To use the **GridSearchCV()** estimator, there are several steps:

- Create an unfit model.

- Create a Python dictionary of the hyperparameters and values you want to try. If the hyperparameter is part of a named step, prepend the hyperparameter with the name of the named step, followed by two underscores.
- Create the GridSearchCV object. Provide the unfit estimator, the parameter you want to try, the metric to be used to score the model, and the training and dev set indices.
- Call the fit() method of the GridSearchCV object. Provide the entire dataset, not just the training set, because this search process requires the DataFrame to carry out simple cross-validation—it needs access to the development set.
- Use the .best_estimator_ property to find the best estimator.
- Use the best_model.predict method to make real-world predictions (as an optional last step).

To access the ridge estimator that was by found by GridSearchCV:

- Ask best_model for the named step
- Use the .coef feature to get an array of numbers
- Convert the numbers into a DataFrame to understand which weights are associated with which features

GridSearchCV also allows you to see the cross-validation scores it computed to make its decision. This is helpful if trying to debug a result. To see the cross-validation results, use the **cv_results_** property.
**model_finder.cv_results_**

It will return a summary with aspects of the fitting process, including:

- **'split0_test_score'**: Gives the MSE of each choice of $\alpha$

- **'mean_test_score'**: Is redundant with split0_test_score for simple cross-validation
- **'rank_test_score'**: Gives the relative rank of each choice of $\alpha$

Warning: Technically, a GridSearchCV object is an estimator, meaning that you can call **model_finder.predict** on some data in an attempt find the best estimator. However, **model_finder.predict** just uses whatever choice of hyperparameters were tried most recently. To find the best model, rather than the most recent model, use **model_finder.best_estimator_**.

A note on comparing models: The process of comparing models is subject to variance. For instance, at the start of an experiment a specific training and dev set was shared by all the models. However, if another training and dev set is generated, the MSEs can be dramatically different.

## LASSO Regression

Rather than use the sum of the squares of the features as the penalty term, you could use the sum of the absolute values. **LASSO regression** is a linear regression model where the penalty term is the sum of the absolute values of the parameters times $\alpha$. It is also called **L1 regularization** because it refers to the L1 norm of the parameters.

Lasso regression is often slower and less numerically stable than ridge regression, but it only uses a relatively small number of features.

$$\alpha \sum_{j=1}^{d} \theta_j^2 + \frac{1}{n} \sum_{i=1}^{n} \left( y_i - \sum_{j=1}^{d} \theta_j \phi_{i,j} \right)^2$$

$$\alpha \sum_{j=1}^{d} |\theta_j| + \frac{1}{n} \sum_{i=1}^{n} \left( y_i - \sum_{j=1}^{d} \theta_j \phi_{i,j} \right)^2$$

LASSO regression in scikit-learn can be used as follows:

- Instantiate a LASSO object
- Call fit(), either directly or by using GridSearchCV()
- Convert the coefficients into a DataFrame

It is common to get convergence warnings when you fit a LASSO model because an L1 regularization results in a harder problem to solve numerically.

When running a grid search using LASSO regression, most of the coefficients are zero, because it forces the parameters to be smaller. But it also forces many of the parameters to zero.

LASSO automatically selects features without a sequential feature selection process. Instead, the zeros emerge organically as part of a global optimization process.

## K-fold Cross-Validation

To perform **simple cross-validation**, you follow this process:

- Split the data into a training and validation set
- Train the model on the training set to determine the quality of a particular hyperparameter
- Evaluate the quality based on the model's error on the validation set

For example, the choices of $\alpha$ can yield three different sets of parameters. In that case, you check the MSE for each parameter set to determine which is the lowest—that is the hyperparameter you pick.

A downside to simple cross-validation is that some of the samples are never used for training and data can be exceedingly hard to come by.

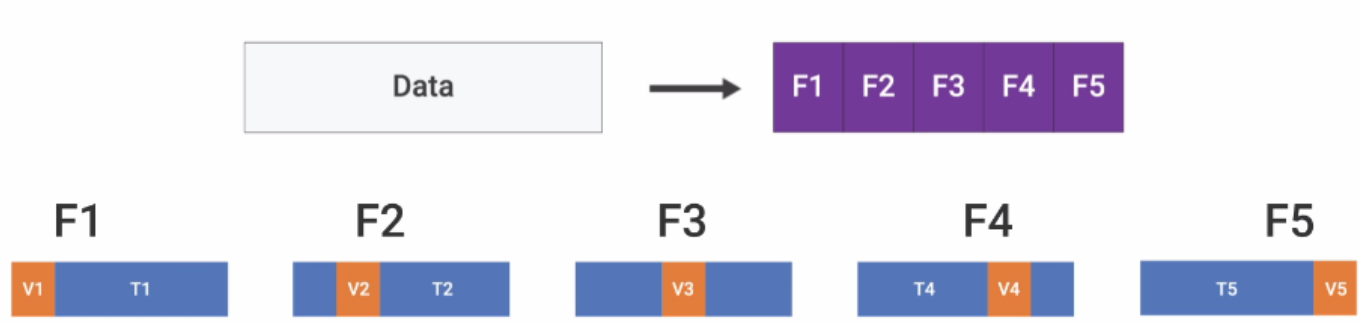
In contrast, **k-fold cross-validation** uses all the data while still avoiding overfitting. However, it is much more computationally expensive.

To perform **k-fold cross-validation**, you would follow this process:

- Split the data into $k$ different groups, often called **folds**—common choices are $k = 5$, $k = 10$, and $k = n$, where $n$ is the number of samples you have
- Pick a fold, known as the **validation fold**, to determine the quality of a particular hyperparameter
- Train the model on all but the validation fold
- Compute the error on the validation fold
- Repeat the process for all $k$ possible choices on the validation fold—the overall quality of the current model is the average of the $k$ different validation fold errors

Where $k = n$, it is also called **leave-one-out** cross-validation. In this approach the temporary validation sets are always exactly one sample and every single datapoint gets a chance to be the validation set. As $k$ increases, the noise in the cross-validation errors tend to decrease. However, the leave-one-out approach is computationally expensive. For example, if there are 10,000 datapoints, then for every hyperparameter you want to evaluate, you need to fit 10,000 models.

Consider this graphical representation of k-fold cross-validation:




In the example figure, for the first iteration, the leftmost fold is used as a validation set. And in the second iteration, the second fold is the temporary validation set. The process is repeated five times, with a different fold used as a validation set each time. To compute the cross-validation error, you average the error computed across each of the five experiments, yielding an overall quality assessment for the choice of hyperparameters under consideration. The best $\alpha$ is the one with the lowest average MSE. To compute the score, five different models need to be fit. In contrast, simple cross-validation only has to fit model to assess a particular choice of hyperparameter.

To perform k-fold cross-validation in scikit-learn is easy. But rather than providing a list of training and dev set indices, you only give a single number as the CV argument. For example, if you set CV = 5 in the constructor of the GridSearchCV object, the resulting search is a five-fold cross-validation.

## Human Resources Applications

As you have seen, there are many tools to build and use predictive models. These tools are designed to forecast and predict outcomes based on the

data provided—that means the **outputs** (predictions) are always a **function of the inputs** (data).

It is important to consider situations where you want outputs that are different from the data used to forecast them, as there may be some benefits from learning or exploration.

For example, designing and implementing algorithms in the context of a human resources decision. Businesses are increasingly looking for ways to automate labor-intensive processes, such as recruiting. When recruiting, companies receive a vast number of applications and may need an automated screen tool to screen applicants based on qualifications.

But what is meant by qualified? The simplest version is to consider qualifications based on inputs—a degree in a relevant field, work experience, et cetera. A more data-driven approach might be to determine, using data and modeling techniques, which applicant characteristics have been traditionally correlated with success among existing employees. However, these models will not be able to predict performance of applicants that look very different from those currently in the workforce. This is not great for workforce diversity. It may also not be great for performance or productivity.

For application screening, you might want a tool that:

- Optimizes hiring based on existing data
- Prioritizes hiring candidates who look different than the existing workforce (for diversity)
- Gathers more data to learn from them so the learning process feeds back into the model for future applicants

This is an example of a **bandit problem**. To find the best workers over time, firms must balance:

- '**Exploitation**'—selecting from groups with proven track records
- '**Exploration**'—selecting from underrepresented groups to learn about quality

A group of economists from the Massachusetts Institute of Technology (MIT) and Columbia University recently built a resume screening algorithm that values exploration by evaluating candidates according to their statistical upside potential—selecting candidates with characteristics for which there is more statistical uncertainty. They observed data on an applicant's demographics, education, and work history. The goal was to maximize the quality of applicants who are selected for an interview.

The researchers built three resume screening algorithms:

- The first algorithm used a **static supervised learning approach** based on a **logit LASSO model**.
- The second algorithm built on the **same baseline as the LASSO model**, but it **updated the training data** throughout the test period. While this updating process allows the LASSO model to learn about the quality of the applicants it selects, it is myopic because it does not incorporate the value of this learning into its selection decisions.
- The third algorithm implemented an **upper-confidence bound, contextual bandit algorithm** and selected applicants based on the upper bound of the confidence interval associated with point estimates**.** There is an exploration bonus that is increasing in the algorithm's degree of uncertainty about quality. Exploration bonuses are higher for groups of candidates who are underrepresented in the

algorithm's training data, because the model has a less precise estimates for these groups.

They evaluated the candidates that each algorithm selected, relative to the actual interview decisions made by the organization's human resume screeners.

There were two main results evident from the paper:

- There is a marked difference in the demographic composition of the applicants selected by the two supervised learning approaches and the contextual bandit approach:
  - 24% black and Hispanic applicants selected for the upper-confidence bound model
  - 5% black and Hispanic applicants selected for the updating LASSO model
  - 2% black and Hispanic applicants selected for the static LASSO model.
- All machine learning models increase hiring yield relative to human recruiters. However, this comparison is not experimental in nature.

This study showed that the upper-confidence bound approach improves hiring quality of selected candidates, while also increasing demographic diversity relative to the firm's existing hiring practices. The same is not true for traditional supervised learning-based algorithms, which improve hiring quality, but select far fewer black and Hispanic applicants.