INTRODUCTION TO ML WITH PYTHON- ANDREAS MUELLER

CHAPTER -1 INTRODUCTION

SciPy

SciPy is a collection of functions for scientific computing in Python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions, and statistical distributions. scikit-learn draws from SciPy's collection of functions for implementing its algorithms. The most important part of SciPy for us is *scipy.sparse*: this provides sparse matrices, which are another representation that is used for data in scikitlearn. Sparse matrices are used whenever we want to store a 2D array that contains mostly zeros.

Sparse Matrix- Matrices that contain mostly zero values are called sparse matrix, distinct from matrices where most of the values are non-zero, called dense.

sparsity = count zero elements / total elements

Sparse matrices come up in encoding schemes used in the preparation of data. Three common examples include:

- **One-hot encoding**, used to represent categorical data as sparse binary vectors.
- **Count encoding**, used to represent the frequency of words in a vocabulary for a document
- **TF-IDF encoding**, used to represent normalized word frequency scores in a vocabulary.

There are also data structures that are more suitable for performing efficient operations; two commonly used examples are listed below.

- Compressed Sparse Row (CSR). The sparse matrix is represented using three one-dimensional arrays for the non-zero values, the extents of the rows, and the column indexes. It's most commonly used. (2,3) 4. Here it means 4 is present in the 3st row of 4th column. Index in python starts with zero.
- Compressed Sparse Column (CSC). The same as the Compressed Sparse Row method except the column indices are compressed and read first before the row indices.

pandas

pandas is a Python library for data wrangling and analysis. It is built around a data structure called the DataFrame that is modeled after the R DataFrame. Simply put, a pandas DataFrame is a table, similar to an Excel spreadsheet. pandas provides a great range of methods to modify and operate on this table; in particular, it allows SQL-like queries and joins of tables. In contrast to NumPy, which requires that all entries in an array be of the same type, pandas allows each column to have a separate type (for example, integers, dates, floating-point numbers, and strings)

CHAPTER - 2 SUPERVISED LEARNING

Overfitting occurs when you fit a model too closely to the particularities of the training set and obtain a model that works well on the training set but is not able to generalize to new data.

Choosing too simple a model which would even do badly on training set is called **underfitting**.

K Nearest Neighbor generally works well if independent features are 100 or lesser and if the data has features mostly filled with 0 (i.e. sparse matrix). Data must be preprocessed. k-NN is quite easier to understand. There are only two parameters to figure out: no. of neighbors to take and how you measure distance b/w datapoints (Euclidean distance is generally used).

Regularization parameter is called alpha in regression and C in SVC and logistic regression. *In sklearn library, L2 regularization is by-default implemented in logistic regression.*

NAIVE BAYES CLASSIFIER:-

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in scikit- learn: *GaussianNB*, *BernoulliNB*, and *MultinomialNB*. GaussianNB can be applied to any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). BernoulliNB and MultinomialNB are mostly used in text data classification.

The BernoulliNB classifier counts how often every feature of each class is not zero. The other two naive Bayes models, MultinomialNB and GaussianNB, are slightly different in what kinds of statistics they compute. MultinomialNB takes into account the average value of each feature for each class, while GaussianNB stores the average value as well as the standard deviation of each feature for each class.

MultinomialNB and BernoulliNB have a single parameter, alpha, which controls model complexity. The way alpha works is that the algorithm adds to the data alpha many virtual data points that have positive values for all the features. This results in a “smoothing” of the statistics. A large alpha means more smoothing, resulting in less complex models. The algorithm’s performance is relatively robust to the setting of alpha, meaning that setting alpha is not critical for good performance. However, tuning it usually improves accuracy somewhat.

GaussianNB is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text. MultinomialNB usually performs better than BinaryNB, particularly on datasets with a relatively large number of nonzero features (i.e., large documents).

DECISION TREE

Decision trees in scikit-learn are implemented in the DecisionTreeRegressor and DecisionTreeClassifier classes. There are two common strategies to prevent overfitting in a decision tree:

- Stopping the creation of the tree early (also called pre-pruning). Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.
- Building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning). scikit-learn only implements pre-pruning, not post-pruning.

STRENGTH AND WEAKNESSES OF DECISION TREE-

Decision trees are quite easier to visualize. As each feature is processed separately, and the possible splits of the data don't depend on scaling, no preprocessing like normalization or standardization of features is needed for decision tree algorithms. In particular, decision trees work well when you have features that are on completely different scales, or a mix of binary and continuous features.

The main downside of decision trees is that even with the use of pre-pruning, they tend to overfit and provide poor generalization performance. Therefore, in most applications, the ensemble methods we discuss next are usually used in place of a single decision tree.

RANDOM FOREST

The idea behind random forests is that each tree might do a relatively good job of predicting but will likely overfit on part of the data. If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

To implement this strategy, we need to build many decision trees. Each tree should do an acceptable job of predicting the target and should also be different from the other trees. Random forests get their name from injecting randomness into the tree building to ensure each tree is different. There are two ways in which the trees in a random forest are randomized: by selecting the data points used to build a tree and by selecting the features in each split test. Let's go into this process in more detail.

Building Random Forest-

TO build a random forest, we need to consider two things:-

- How many trees to build (*n_estimators* in sklearn). Take the bootstrap sample of trees with sampling. At a time, take 75% of the data from this particular sample data and predict on 25% and get OOB error/misclassification. This sample data is called *n_sample* in sklearn.
- How many features to search over to make split on a feature(*max_features* in sklearn)

GRADIENT BOOSTED REGRESSION TREES (GRADIENT BOOSTING MACHINES)

In contrast to the random forest approach, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one.

The main idea behind gradient boosting is to combine many simple models (in this context known as weak learners), like shallow trees. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance.

Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the learning_rate, which controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make stronger corrections, allowing for more complex models.

Adding more trees to the ensemble, which can be accomplished by increasing n_estimators, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

These two parameters are highly interconnected, as a lower learning_rate means that more trees are needed to build a model of similar complexity. In contrast to random forests, where a higher n_estimators value is always better, increasing n_estimators in gradient boosting leads to a more complex model, which may lead to overfitting. A common practice is to fit n_estimators depending on the time and memory budget, and then search over different learning_rates.

Another important parameter is max_depth (or alternatively max_leaf_nodes), to reduce the complexity of each tree. Usually max_depth is set very low for gradient boosted models, often not deeper than five splits.

KERNEL SUPPORT VECTOR MACHINE

During training, the SVM learns how important each of the training data points is torepresent the decision boundary between the two classes. Typically only a subset of the training points matter for defining the decision boundary: the ones that lie on the border between the classes. These are called support vectors and give the support vector machine its name.

The important parameters in kernel SVMs are the regularization parameter C, the choice of the kernel, and the kernel-specific parameters. Although we primarily focused on the RBF kernel, other choices are available in scikit-learn. The RBF kernel has only one parameter, gamma, which is the inverse of the width of the Gaussian kernel. gamma and C both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and C and gamma should be adjusted together.

Tuning SVM parameters:- The gamma parameter is the one shown in the formula given in the previous section, which controls the width of the Gaussian kernel. It determines the scale of what it means for points to be close together. The C parameter is a regularization parameter, similar to that used in the linear models. It limits the importance of each point (or more precisely, their dual_coef_).

Preprocessing data for SVMs:- If input features are in different order of magnitude (Say age in 10s while salaries in 000's), then SVM would give worse results. One way to resolve this problem is by rescaling each feature so that they are all approximately on the same scale. A common rescaling method for kernel SVMs is to scale the data such that all features are between 0 and 1. We will see how to do this using the MinMaxScaler preprocessing method in Chapter 3, where we'll give more details.

it might be worth trying SVMs, particularly if all of your features represent measurements in similar units (e.g., all are pixel intensities) and they are on similar scales.

The important parameters in kernel SVMs are the regularization parameter C, the choice of the kernel, and the kernel-specific parameters. Although we primarily focused on the RBF kernel, other choices are available in scikit-learn. The RBF kernel has only one parameter, gamma, which is the inverse of the width of the Gaussian kernel. gamma and C both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and C and gamma should be adjusted together.

NEURAL NETWORK

Here, we will only discuss some relatively simple methods of NN, namely multilayer perceptrons for classification and regression, that can serve as a starting point for more involved deep learning methods. Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks.

MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.

This model has a lot more coefficients (also called weights) to learn: there is one between every input and every hidden unit (which make up the hidden layer), and one between every unit in the hidden layer and the output.

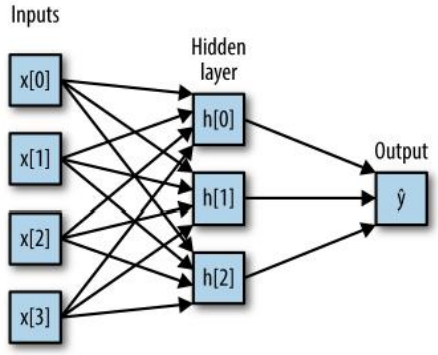


Figure 2-45. Illustration of a multilayer perceptron with a single hidden layer

Computing a series of weighted sums is mathematically the same as computing just one weighted sum, so to make this model truly more powerful than a linear model, we need one extra trick. After computing a weighted sum for each hidden unit, a nonlinear function is applied to the result—usually the rectifying nonlinearity (also known as rectified linear unit or relu) or the tangens hyperbolicus (tanh). The result of this function is then used in the weighted sum that computes the output, ŷ.

The relu cuts off values below zero, while tanh saturates to –1 for low input values and +1 for high input values. Either nonlinear function allows the neural network to learn much more complicated functions than a linear model could.

Neural Network also requires data to be rescaled; just like kernel SVM.

If you are interested in working with more flexible or larger models, we encourage you to look beyond scikit-learn into the fantastic deep learning libraries that are out there. For Python users, the most well-established are keras, lasagna, and tensor-flow. lasagna builds on the theano library, while keras can use either tensor-flow or theano. These libraries provide a much more flexible interface to build neural networks and track the rapid progress in deep learning research. All of the popular deep learning libraries also allow the use of highperformance graphics processing units (GPUs), which scikit-learn does not support. Using GPUs allows us to accelerate computations by factors of 10x to 100x, and they are essential for applying deep learning methods to large-scale datasets.

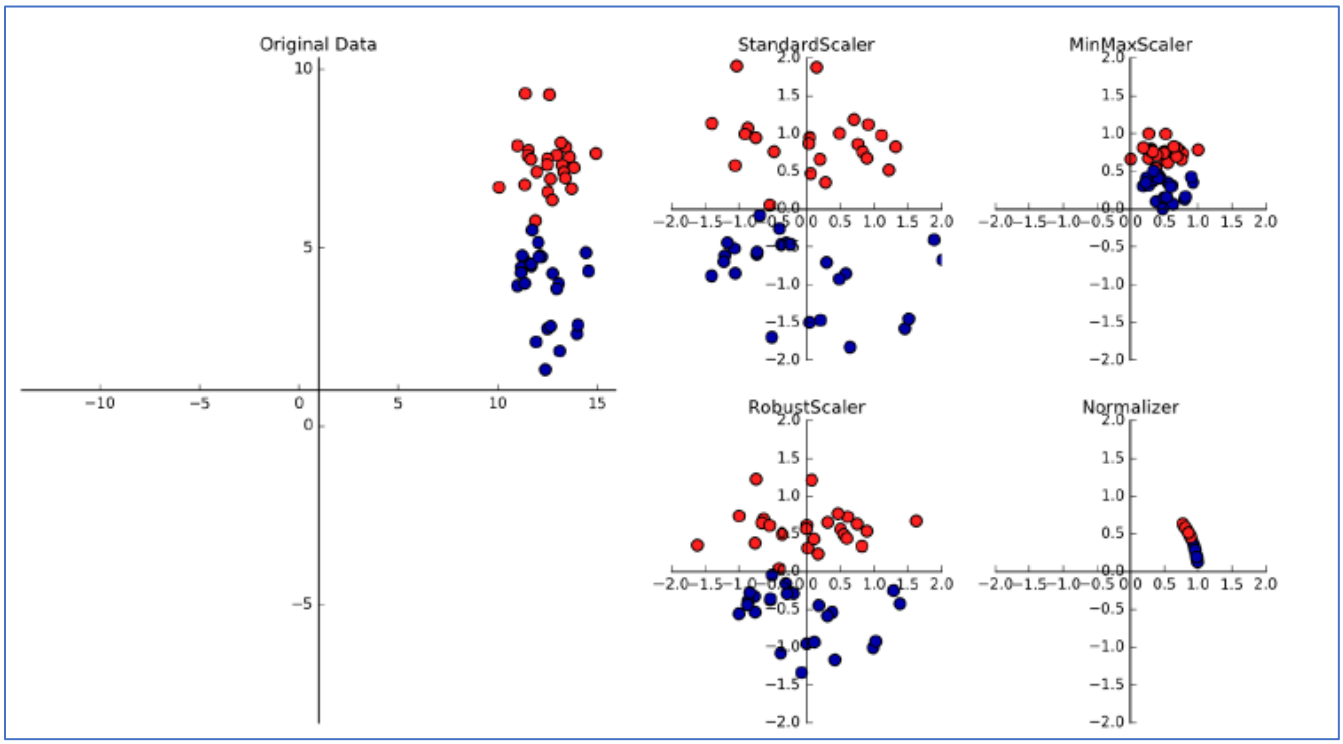
Estimating complexity in neural networks- The most important parameters are the number of layers and the number of hidden units per layer. You should start with one or two hidden layers, and possibly expand from there. The number of nodes per hidden layer is often similar to the number of input features, but rarely higher than in the low to mid-thousands.

A helpful measure when thinking about the model complexity of a neural network is the number of weights or coefficients that are learned. If you have a binary classification dataset with 100 features, and you have 100 hidden units, then there are 100 * 100 = 10,000 weights between the input and the first hidden layer. There are also 100 * 1 = 100 weights between the hidden layer and the output layer, for a total of around 10,100 weights. If you add a second hidden layer with 100 hidden units, there will be another 100 * 100 = 10,000 weights from the first hidden layer to the second hidden layer, resulting in a total of 20,100 weights. If instead you use one layer with 1,000 hidden units, you are learning 100 * 1,000 = 100,000 weights from the input to the hidden layer and 1,000 x 1 weights from the hidden layer to the output layer, for a total of 101,000. If you add a second hidden layer you add 1,000 * 1,000 = 1,000,000 weights, for a whopping total of 1,101,000—50 times larger than the model with two hidden layers of size 100.

A common way to adjust parameters in a neural network is to first create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Then, once you know the training data can be learned, either shrink the network or increase alpha to add regularization, which will improve generalization performance.

A model is called calibrated if the reported uncertainty actually matches how correct it is–in a **calibrated** model, a prediction made with 70% certainty would be correct 70% of the time.

CHAPTER 3- UNSUPERVISED LEARNING



The first plot in above figure shows a synthetic two-class classification dataset with two features. The first feature (the x-axis value) is between 10 and 15. The second feature (the y-axis value) is between around 1 and 9.

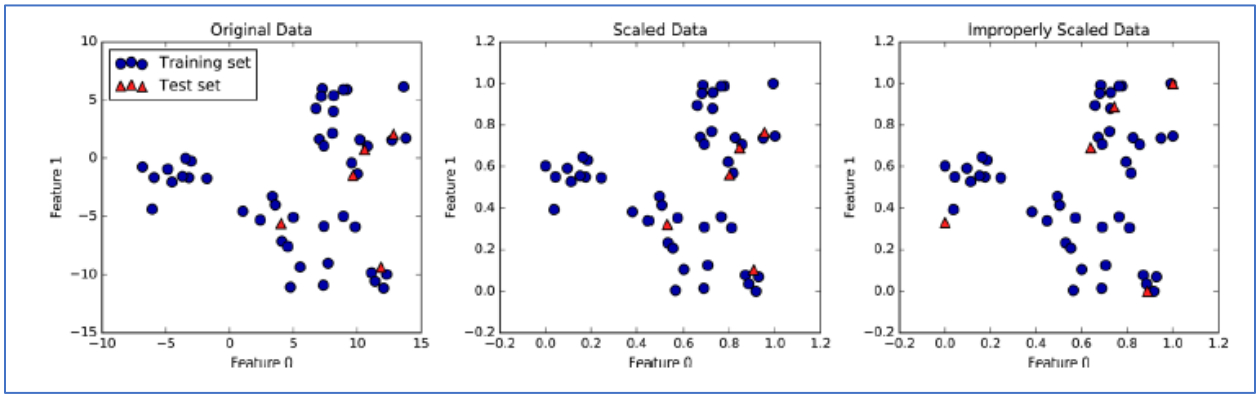
The above four plots show four different ways to transform the data that yield more standard ranges.

- The **StandardScaler** in scikit-learn ensures that for each feature the mean is 0 and the variance is 1, bringing all features to the same magnitude. However, this scaling does not ensure any particular minimum and maximum values for the features.
- The **RobustScaler** works similarly to the StandardScaler in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the RobustScaler uses the median and quartiles, instead of mean and variance. This makes the RobustScaler ignore data points that are very different from the rest (like measurement errors). These odd data points are also called outliers, and can lead to trouble for other scaling techniques.
- The **MinMaxScaler**, on the other hand, shifts the data such that all features are exactly between 0 and 1. For the two-dimensional dataset this means all of the data is contained within the rectangle created by the x-axis between 0 and 1 and the y-axis between 0 and 1.
- Finally, the **Normalizer** does a very different kind of rescaling. It scales each data point such that the feature vector has a Euclidean length of 1. In other words, it projects a data point on the circle (or sphere, in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of its length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

Scaling Training and Test Data the Same Way

MinMaxScaler and all the other scalers always apply exactly the same transformation to the training and the test set. This means the transform method always subtracts the training set minimum and divides by the training set range, which might be different from the minimum and range for the test set. For test set, minimum and range is not used from test but from training.

The following figure illustrates what would happen if we were to use the minimum and range of the test set instead. In figure 3, we have used the min and range of test to transform test set (we use MinMaxScaler for demonstration here). We can see that that test set observations in the plot 3 have changed their location. While in figure 2, they are in the same relative position to training set.



Dimensionality Reduction, Feature Extraction, and Manifold Learning

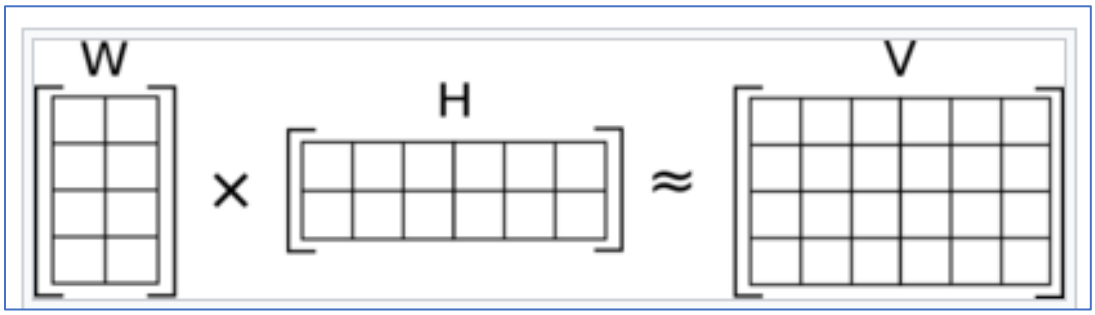
One of the simplest and most widely used algorithms for all of these is principal component analysis. We'll also look at two other algorithms: non-negative matrix factorization (NMF), which is commonly used for feature extraction, and t-SNE, which is commonly used for visualization using two-dimensional scatter plots.

Principal Component Analysis (PCA):- Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data. The 1st component is along the feature whose variance is highest. 2nd component is always orthogonal on 1st component.

Non-Negative Matrix Factorization (NMF):- An unsupervised method in linear algebra and clustering where a matrix V is factorized into (usually) two matrices W and H, with the property that all three matrices have no negative elements. This non-negativity makes the resulting matrices easier to inspect.

NMF finds applications in such fields as computer vision, document clustering, **audio signal processing** etc.

Here the matrix V is represented by the two smaller matrices W and H, which, when multiplied, approximately reconstruct V.



Look at user guide of sklearn to check more unsupervised methods like NMF and PCA.

http://scikit-learn.org/stable/unsupervised_learning.html

Manifold Learning with t-SNE:- There is a class of algorithms for visualization called manifold learning algorithms that allow for much more complex mappings, and often provide better visualizations. A particularly useful one is the t-SNE algorithm.

Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to generate more than two new features. Some of them, including t-SNE, compute a new representation of the training data, but don't allow transformations of new data. This means these algorithms cannot be applied to a test set: rather, they can only transform the data they were trained for. Manifold learning can be useful for exploratory data analysis, but is rarely used if the final goal is supervised learning. The idea behind t-SNE is to find a two-dimensional representation of the data that preserves the distances between points as best as possible. t-SNE starts with a random two-dimensional representation for each data point, and then tries to make points that are close in the original feature space closer, and points that are far apart in the original feature space farther apart. t-SNE puts more emphasis on points that are close by, rather than preserving distances between far-apart points. In other words, it tries to preserve the information indicating which points are neighbors to each other.

We already know the in and outs of k-means and hierarchical, hence wont discuss it here. Only DBSCAN would be discussed.

DBSCAN:- Another very useful clustering algorithm is DBSCAN (density-based spatial clustering of applications with noise). The main benefits of DBSCAN are that it does not require the user to set the number of clusters a priori, it can capture clusters of complex shapes, and it can identify points that are not part of any cluster.

DBSCAN works by identifying points that are in "crowded" regions of the feature space, where many data points are close together. These regions are referred to as dense regions in feature space. The idea behind DBSCAN is that clusters form dense regions of data, separated by regions that are relatively empty.

Points that are within a dense region are called core samples (or core points), and they are defined as follows. There are two parameters in DBSCAN in sklearn: min_samples and eps. If there are at least

min_samples many data points within a distance of eps to a given data point, that data point is classified as a core sample. Core samples that are closer to each other than the distance eps are put into the same cluster by DBSCAN. The algorithm works by picking an arbitrary point to start with. It then finds all points with distance eps or less from that point. If there are less than min_samples points within distance eps of the starting point, this point is labeled as noise, meaningthat it doesn't belong to any cluster. If there are more than min_samples points within a distance of eps, the point is labeled a core sample and assigned a new cluster label.

Then, all neighbors (within eps) of the point are visited. If they have not been assigned a cluster yet, they are assigned the new cluster label that was just created. If they are core samples, their neighbors are visited in turn, and so on. The cluster grows until there are no more core samples within distance eps of the cluster. Then another point that hasn't yet been visited is picked, and the same procedure is repeated.

In the end, there are three kinds of points: core points, points that are within distance eps of core points (called boundary points), and noise. When the DBSCAN algorithm is run on a particular dataset multiple times, the clustering of the core points is always the same, and the same points will always be labeled as noise. However, a boundary point might be neighbor to core samples of more than one cluster. Therefore, the cluster membership of boundary points depends on the order in which points are visited. Usually there are only few boundary points, and this slight dependence on the order of points is not important.

Summary of Algorithms:-

k-means allows for a characterization of the clusters using the cluster means. It can also be viewed as a decomposition method, where each data point is represented by its cluster center. DBSCAN allows for the detection of "noise points" that are not assigned any cluster, and it can help automatically determine the number of clusters.

Agglomerative clustering can provide a whole hierarchy of possible partitions of the data, which can be easily inspected via dendrograms.

CHAPTER 4- REPRESENTING DATA AND FEATURE ENGINEERING

One-Hot-Encoding (Dummy Variables)

By far the most common way to represent categorical variables is using the one-hot-encoding or one-out-of-N encoding, also known as dummy variables. The idea behind dummy variables is to replace a categorical variable with one or more new features that can have the values 0 and 1. There are two ways to convert your data to a one-hot encoding of categorical variables, using either pandas or scikit-learn.

If there is a numeric variable, we can use pandas to convert it to categorical but first forcing this variable into str type then use get_dummies().

We call get_dummies on a DataFrame which is a combination of both the training and the test data.

Imagine the workclass feature has the values "Government Employee" and "Private Employee" in the training set, and "Self Employed" and "Self Employed Incorporated" in the test set. In both cases, pandas will create two new dummy features, so the encoded Data Frames will have the same number of features. However, the two dummy features have entirely different meanings in the training and test sets. The column that means "Government Employee" for the training set would encode "Self Employed" for the test set. If we built a machine learning model on this data it would work very badly, because it would assume the columns mean the same things (because they are in the same index position) when in fact they mean very different things.

If some features have a non-linear relationship with response variable and we wana do regression. Then we can bin these numeric independent variables for better results on linear regression.

Univariate Non-linear Transformations:-

http://scikit-learn.org/stable/modules/feature_selection.html

Adding squared or cubed features can help linear models for regression. There are other transformations that often prove useful for transforming certain features: in particular, applying mathematical functions like *log, exp, or sin*. While tree-based models only care about the ordering of the features, linear models and neural networks are very tied to the scale and distribution of each feature, and if there is a nonlinear relation between the feature and the target, that becomes hard to model — particularly in regression. Most models work best when each feature (and in regression also the target) is loosely Gaussian distributed (normal curve shape). A particularly common case when such a transformation can be helpful is when dealing with integer count data. Finding the transformation that works best for each combination of dataset and model is somewhat of an art. In this example, all the features had the same properties. This is rarely the case in practice, and usually only a subset of the features should be transformed, or sometimes each feature needs to be transformed in a different way. As we mentioned earlier, these kinds of transformations are irrelevant for tree-based models but might be essential for linear models. Sometimes it is also a good idea to transform the target variable y in regression. Trying to predict counts (say, number of orders) is a fairly common task, and using the log(y + 1) transformation often helps.

Automatic Feature Selection

When building a model, it is always a good idea to use only features which generalize the model better. There are three basic strategies:

- univariate statistics
- model-based selection and
- iterative selection

Univariate Statistics

In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as analysis of variance (ANOVA).

To use univariate feature selection in scikit-learn, you need to choose a test, usually either f_classif (the default) for classification or f_regression for regression, and a method to discard features based on the p-values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a p-value (which means they are unlikely to be related to the target). The methods differ in how they compute this threshold, with the simplest ones being **SelectKBest**, which selects a fixed number k of features, and **SelectPercentile**, which selects a fixed percentage of features.

Model Based Selection:-

Model-based feature selection uses a supervised machine learning model to judge the importance of each feature and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling. The feature selection model needs to provide some measure of importance for each feature, so that they can be ranked by this measure.

To use model-based feature selection, we need to use the *SelectFromModel* transformer. Generally, classification algos have *feature_importances_* while regression algos have *coef_* attributes after fitting the model. The features are considered unimportant and removed, if the corresponding `coef_` or `feature_importances_` values are below the provided `threshold` parameter. Apart from specifying the threshold numerically, there are built-in heuristics for finding a threshold using a string argument. Available heuristics are "mean", "median" and float multiples of these like "0.1*mean"

Iterative Selection:-

In univariate testing we used no model, while in model-based selection we used a single model to select features. In iterative feature selection, a series of models are built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one until some stopping criterion is reached, or starting with all features and removing features one by one until some stopping criterion is reached. One particular method of this kind is **recursive feature elimination (RFE)**, which starts with all features, builds a model, and discards the least important feature according to the model. Then a new model is built using all but the discarded feature, and so on until only a prespecified number of features are left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model-based selection.

CHAPTER 5- MODEL EVALUATION AND IMPROVEMENT

So far, we have split data into train & test set, trained model on train and tested accuracy on test set. We do this to check the generalization of model on unknown data. We can check the generalization using two more robust methods in sklearn:-

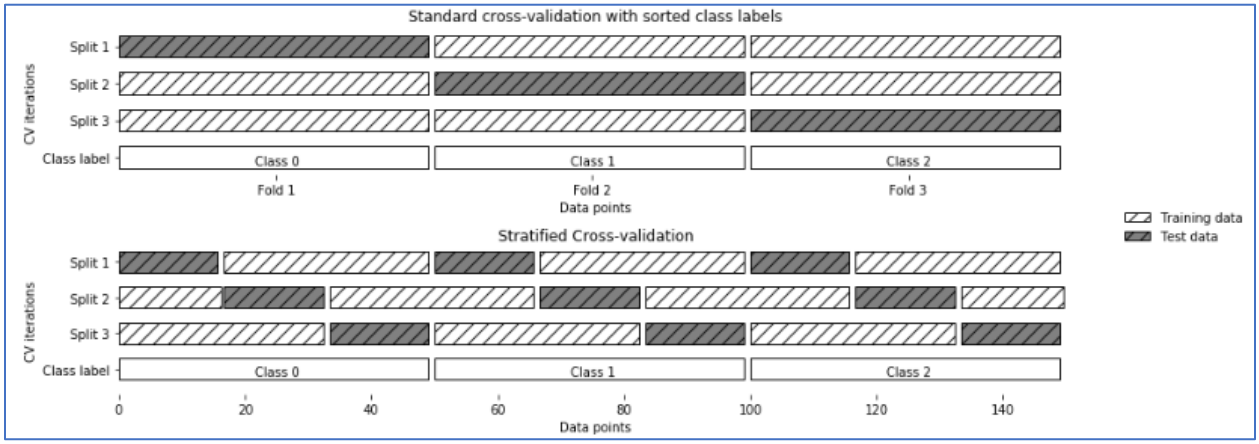
- Cross Validation
- Grid Search

Cross-validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and a test set. In cross validation, the data is instead split repeatedly and multiple models are trained. The most commonly used version of cross-validation is k-fold cross-validation, where k is a user-specified number, usually 5 or 10. When performing five-fold cross-validation, the data is first partitioned into five parts of (approximately) equal size, called folds. Next, a sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds (2–5) are used as the training set. The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1. Then another model is built, this time using fold 2 as the test set and the data in folds 1, 3, 4, and 5 as the training set. This process is repeated using folds 3, 4, and 5 as test sets. For each of these five splits of the data into training and test sets, we compute the accuracy. In the end, we have collected five accuracy values.

Another benefit of cross-validation as compared to using a single split of the data is that we use our data more effectively. When using train_test_split, we usually use 75% of the data for training and 25% of the data for evaluation. When using five-fold cross-validation, in each iteration we can use four-fifths of the data (80%) to fit the model. When using 10-fold cross-validation, we can use nine-tenths of the data (90%) to fit the model. More data will usually result in more accurate models.

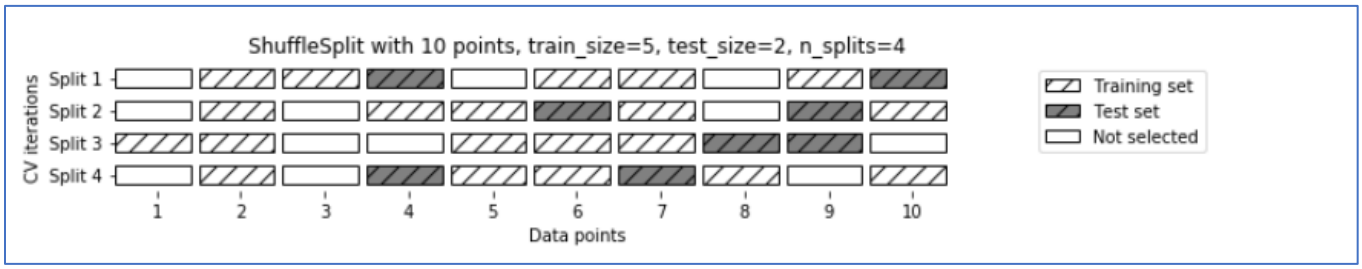
Stratified k-Fold Cross-Validation and Other Strategies

In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated in Figure



Shuffle-Split Cross-Validation:-

Another method is shuffle-split cross-validation where some folds are in train set, others in test set and some fold aren't selected. This is repeated n_iter times. Here dataset has 10 points. We have selected 5 points for train and 2 for test.



Shuffle-split cross-validation allows for control over the number of iterations independently of the training and test sizes, which can sometimes be helpful. It also allows for using only part of the data in each iteration, by providing train_size and test_size settings that don't add up to one. Subsampling the data in this way can be useful for experimenting with large datasets.

There is also a stratified variant of ShuffleSplit, aptly named *StratifiedShuffleSplit*, which can provide more reliable results for classification tasks.

Cross-validation with groups

GroupKFold, which takes an array of groups as argument that we can use to indicate which person/object is in the image/data. The groups array here indicates groups in the data that should not be split when creating the training and test sets, and should not be confused with the class label.

This example of groups in the data is common in medical applications, where you might have multiple samples from the same patient, but are interested in generalizing to new patients. Similarly, in speech recognition, you might have multiple recordings of the same speaker in your dataset, but are interested in recognizing speech of new speakers.

There are more splitting strategies for cross-validation in scikit-learn, which allow for an even greater variety of use cases but the standard KFold, StratifiedKFold, and GroupKFold are by far the most commonly used ones. You can check the rest of methods here:

http://scikit-learn.org/stable/modules/cross_validation.html

GRID SEARCH

Finding the values of important parameters is quite important to make a model generalize. For example, SVM with RBF have two paramaters; alpha & C. We can take diff values of alpha& C and check which ones best for our model. This process is called Grid Search.

Say we want to try the values 0.001, 0.01, 0.1, 1, 10, and 100 for the parameter C, and the same for gamma. Because we have six different settings for C and gamma that we want to try, we have 36

combinations of parameters in total. Looking at all possible combinations creates a table (or grid) of parameter settings for the SVM, as shown here:

	C = 0.001	C = 0.01	...	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	...	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	...	SVC(C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	...	SVC(C=10, gamma=100)

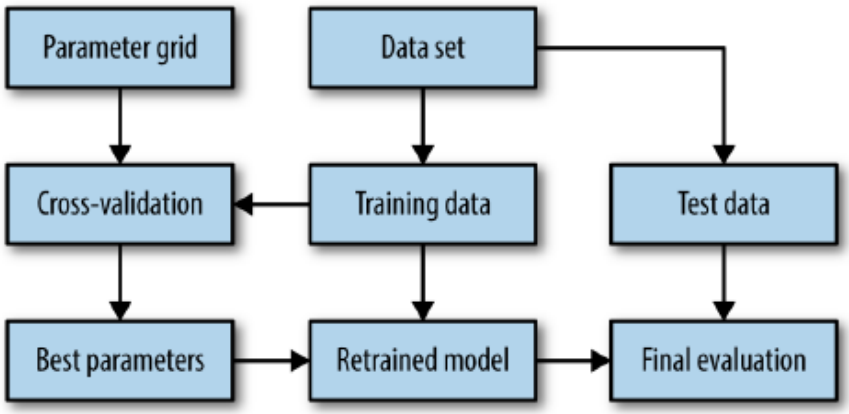
Grid Search With Cross Validation

While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split.

For a better estimate of the generalization performance, instead of using a single split into a training and a validation set, we can use cross-validation to evaluate the performance of each parameter combination.

The overall process of splitting the data, running the grid search, and evaluating the final parameters is illustrated in Figure

```
mglearn.plots.plot_grid_search_overview()
```



Because grid search with cross-validation is such a commonly used method to adjust parameters, scikit-learn provides the GridSearchCV class, which implements it in the form of an estimator. To use the GridSearchCV class, you first need to specify the parameters you want to search over using a dictionary. GridSearchCV will then perform all the necessary model fits. AFter creating the dictionary of parameters for GridSearchCV, then we We can now instantiate the GridSearchCV class with the model (SVC), the parameter grid to search (param_grid), and the cross-validation strategy we want to use (say, five-fold stratified cross-validation).

GridSearchCV will use cross-validation in place of the split into a training and validation set that we used before. However, we still need to split the data into a training and a test set, to avoid overfitting the parameters.

The grid_search object that we created behaves just like a classifier; we can call the standard methods fit, predict, and score on it.1 However, when we call fit, it will run cross-validation for each combination of parameters we specified in param_grid.

Results of trained model using cv are stored in *cv_results_* in the model we trained. If our model is grid_search, we can access the results of diff parameters using *grid_search.cv_results_*

Search over spaces that are not grids

In some cases, trying all possible combinations of all parameters as GridSearchCV usually does, is not a good idea. For example, in svm, we need both C & gamma value when use 'rbf' kernel. But when kernel='linear', we need only C & not gamma.

Nested Cross-Validation

In the preceding examples, we went from using a single split of the data into training, validation, and test sets to splitting the data into training and test sets and then performing cross-validation on the training set. But when using GridSearchCV as described earlier, we still have a single split of the data into training and test sets, which might make our results unstable and make us depend too much on this single split of the data. We can go a step further, and instead of splitting the original data into training and test sets once, use multiple splits of cross-validation. This will result in what is called nested cross-validation. In nested cross-validation, there is an outer loop over splits of the data into training and test sets. For each of them, a grid search is run (which might result in different best parameters for each split in the outer loop). Then, for each outer split, the test set score using the best settings is reported.

The result of this procedure is a list of scores—not a model, and not a parameter setting. The scores tell us how well a model generalizes, given the best parameters found by the grid. As it doesn't provide a model that can be used on new data, nested cross-validation is rarely used when looking for a predictive model to apply to future data. However, it can be useful for evaluating how well a given model works on a particular dataset.

EVALUATION METRICS AND SCORING

$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Recall} = \frac{TP}{TP+FN}$$

So, while precision and recall are very important measures, looking at only one of them will not provide you with the full picture. One way to summarize them is the f-score or f-measure, which is with the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

As it takes precision and recall into account, it can be a better measure than accuracy on imbalanced binary classification datasets.

Increasing the threshold (t of predicted probability) means that the model needs to be more confident to make a positive decision (and less confident to make a negative decision). While working with probabilities may be more intuitive than working with arbitrary thresholds, not all models provide realistic models of uncertainty (a DecisionTree that is grown to its full depth is always 100% sure of its decisions, even though it might often be wrong). This relates to the concept of calibration: a calibrated model is a model that provides an accurate measure of its uncertainty.

Metrics for Multi-Class Classification

The most commonly used metric for imbalanced datasets in the multiclass setting is the multiclass version of the f-score. The idea behind the multiclass f-score is to compute one binary f-score per class, with that class being the positive class and the other classes making up the negative classes. Then, these per-class f-scores are averaged using one of the following strategies:

- "macro" averaging computes the unweighted per-class f-scores. This gives equal weight to all classes, no matter what their size is.
- "weighted" averaging computes the mean of the per-class f-scores, weighted by their support. This is what is reported in the classification report.
- "micro" averaging computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and fscore using these counts.

If you care about each sample equally much, it is recommended to use the "micro" average f1-score; if you care about each class equally much, it is recommended to use the "macro" average f1-score

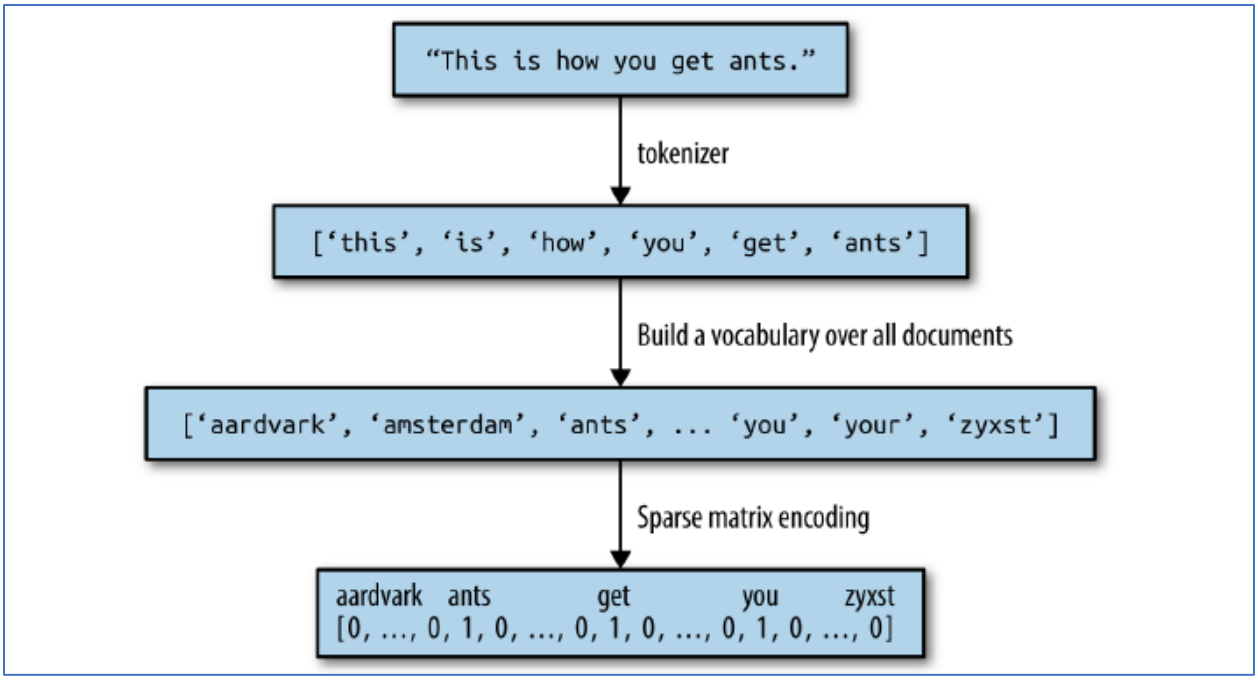
CHAPTER- 6 ALGORITHM CHAINS AND PIPELINES

Nothing to see here. Check Github bruh!!

CHAPTER- 7 WORKING WITH TEXT DATA

Computing the bag-of-words representation for a corpus of documents consists of the following three steps:

1. **Tokenization**- Split each document into the words that appear in it (called tokens), for example by splitting them on whitespace and punctuation.
2. **Vocabulary building**- Collect a vocabulary of all words that appear in any of the documents, and number them (say, in alphabetical order).
3. **Encoding**- For each document, count how often each of the words in the vocabulary appear in this document



TF-IDF (Term Frequency- Inverse Document Frequency)

TF-IDF is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

scikit-learn implements the tf–idf method in two classes: TfidfTransformer, which takes in the sparse matrix output produced by CountVectorizer and transforms it, and TfidfVectorizer, which takes in the text data and does both the bag-of-words feature extraction and the tf–idf transformation.

TF (Term Frequency)- It measures how frequently a term occurs in a document. TF = (Number of times term t appears in a document) / (Total number of terms in the document)

IDF (Inverse Document Frequency)- It measures how important a term is in the whole corpus. IDF = IDF(t) = log_e(Total number of documents / Number of documents with term t in it)

For example, when a 100-word document contains the term “cat” 12 times, the TF for the word ‘cat’ is: TFcat = 12/100 i.e. 0.12

Let's say there are 1000 documents in the corpus. Let’s assume there are 30 documents that contain the term “cat”, then the IDF = log10(1000/30) = 1.52

so, TF-IDF for Cat would be; Wcat= TF * IDF = 0.12 * 1.52 = 0.182

N-Gram in Text Mining

They are basically a set of co-occuring words within a given window and when computing the n-grams you typically move one word forward (although you can move X words forward in more advanced scenarios). For example, for the sentence "The cow jumps over the moon".

If N=2 (known as bigrams), then the ngrams would be:

- the cow
- cow jumps
- jumps over
- over the
- the moon

So you have 5 n-grams in this case. Notice that we moved from the->cow to cow->jumps to jumps->over, etc, essentially moving one word forward to generate the next bigram.

If N=3, the n-grams (trigram) would be:

- the cow jumps
- cow jumps over
- jumps over the
- over the moon

If X=Num of words in a given sentence K, the number of n-grams for sentence K would be:

$$Ngrams_K = X - (N - 1)$$