Pages 1-10

* Install anaconda package
* Anaconda comes with NumPy, SciPy, matplotlib, pandas, IPython, Jupyter Notebook, and scikit-learn.
* scikit-learn is built on top of the NumPy and SciPy scientific Python libraries.
* The Jupyter Notebook is an interactive environment for running code in the browser. It is a great tool for exploratory data analysis and is widely used by data scientists.
* NumPy is one of the fundamental packages for scientific computing in Python. It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudorandom number generators.
* 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.
* (Scipy) sparse.csr\_matrix(input matrix)
* Sparse.coo\_matrix(input matrix)
* matplotlib is the primary scientific plotting library in Python.
* 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.
* Run (terminal) “pydoc modules” to list all python packages.
* Run (terminal) “pip3 install \*\*\*\*”
* .format(x)
* np.array(), np.ones(), np.zeros()
* import matplotlib.pyplot as plt, x=np.linspace(-10,10,1000), plt.plot(x,y,marker="x"), plt.show()
* data={"amin":[1,2,3],"khoshkenar":["a","b","c"]}, pd.DataFrame(data), data\_pandas[data\_pandas.amin>1]

pages 11-20

* import sys, print(sys.version), import numpy as np, print(np.\_\_version\_\_)
* using a test set containing 25% of the data is a good rule of thumb.
* from sklearn.datasets import load\_iris, iris\_dataset=load\_iris()
* print(iris\_dataset.keys()), #print(iris\_dataset.values())
* print(type(iris\_dataset['data'])), print(iris\_dataset['data'].shape)
* from sklearn.model\_selection import train\_test\_split
* X\_train, X\_test, y\_train, y\_test=train\_test\_split(iris\_dataset['data'],iris\_dataset['target'], random\_state=0), random\_state is the seed
* print(X\_train.shape)
* iris\_dataframe=pd.DataFrame(X\_train, columns=iris\_dataset.feature\_names)
* pd.plotting.scatter\_matrix(iris\_dataframe, c=y\_train, figsize=(10,10), marker="o", hist\_kwds={'bins':20}, alpha=0.8)

pages 21-30

* There are two major types of supervised machine learning problems, called *classification* and *regression*.
* Choosing too simple a model is called *underfitting*.
* Building a model that is too complex for the amount of information we have is called *overfitting*.
* from sklearn.neighbors import KNeighborsClassifier
* knn=KNeighborsClassifier(n\_neighbors=1), knn.fit(X\_train,y\_train)
* X\_new=np.array([[5,2.9,1,0.2]])
* prediction=knn.predict(X\_new), np.mean(y\_pred==y\_test), knn.score(X\_test,y\_test)

Pages 31-40

* In knn, for each test point, we count how many neighbors belong to class 0 and how many neighbors belong to class 1. k shows how many neighbors we should search for.
* sudo pip3 install mglearn
* print(np.bincount(cancer.target))
* cancer\_df[['mean radius','mean texture']]
* cancer\_df.loc[1:3,['mean radius','mean texture']]
* cancer\_df.iloc[1:5,[1,4]] (brings 1-4, 1-3)
* cancer\_df.head(), cancer\_df.tail()
* cancer\_df.at[1,'mean radius'], cancer\_df.iat[1,0]
* cancer\_df.columns
* cancer\_df.size

pages 41-50

* knn regression (normalize predictors and calculate Euclidean distance then choose k smallest distances and take average over their corresponding targets.)
* there are two important parameters to the KNeighbors classifier: the number of neighbors and how you measure distance between data points.
* knn approach often does not perform well on datasets with many features (hundreds or more), and it does particularly badly with datasets where most features are 0 most of the time (so-called *sparse datasets*).
* Training set score: 0.95, Test set score: 0.61. This discrepancy between performance on the training set and the test set is a clear sign of overfitting
* range(1,11)
* training\_accuracy=[], training\_accuracy.append()
* X,y=mglearn.datasets.make\_wave(n\_samples=40)
* from sklearn.linear\_model import LinearRegression
* lr=LinearRegression().fit(X\_train,y\_train), print(lr.coef\_), print(lr.intercept\_)
* print(lr.score(X\_train,y\_train)) (this is R2)

pages 51-60

* Regularization means explicitly restricting a model to avoid overfitting.
* The Ridge model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set.
* The lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization. The consequence of L1 regularization is that when using the lasso, some coefficients are *exactly zero*.
* In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice.
* (Linear model of classification) If the function is smaller than zero, we predict the class –1; if it is larger than zero, we predict the class +1.
* For linear models for regression, the output, *ŷ*, is a linear function of the features: a line, plane, or hyperplane (in higher dimensions). For linear models for classification, the *decision boundary* is a linear function of the input.
* from sklearn.linear\_model import Ridge, Ridge(alpha=0.10).fit(X\_train, y\_train)
* from sklearn.linear\_model import Lasso
* Lasso(alpha=0.01,max\_iter=100000).fit(X\_train,y\_train)
* from sklearn.linear\_model import LogisticRegression
* from sklearn.svm import LinearSVC
* plt.subplot(1, 2, 1)
* zip

pages 61-70

* If training and test set performance are very close, it is likely that we are underfitting.
* higher values of C correspond to *less* regularization (in Logistic Regression and SVC).
* The default value of C is 1. LogisticRegression applies an L2 regularization by default
* len(cancer.feature\_names)
* A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the *one-vs.-rest* approach. In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes. To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class “wins,” and this class label is returned as the prediction.
* The mathematics behind multiclass logistic regression differ somewhat from the one- vs.-rest approach.
* But what about the triangle in the middle of the plot? All three binary classifiers classify points there as “rest.” Which class would a point there be assigned to? The answer is the one with the highest value for the classification formula: the class of the closest line.
* Linear models often perform well when the number of features is large compared to the number of samples. However, in lower-dimensional spaces, other models might yield better generalization performance.
* How to run selected code in IDLE python?
* logreg100=LogisticRegression(C=100).fit(X\_train,y\_train)
* plt.xticks(range(cancer.data.shape[1]), cancer.feature\_names, rotation=90)
* for svm plotting -> plt.plot(line,-(line\*coef[0]+intercept)/coef[1])
* for coef, intercept in zip(linear\_svm.coef\_, linear\_svm.intercept\_):

pages 71-80

* Naïve Bayes does not consider order, that’s why it is called naïve.
* In naïve Bayes, calculate the probability of each class (prior probability), and in each class calculate the probability of instances. Then calculate the probability of each group and choose a group with higher probability.
* naive Bayes models often provide generalization performance that is slightly worse than that of linear classifiers like LogisticRegression and LinearSVC. But their training is faster.
* 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.
* GaussianNB, BernoulliNB, and MultinomialNB in scikit-learn.
* 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.
* Pages 71-72, read again!
* In decision tree, a prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target (or the single target in the case of pure leaves) in that region.
* Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data.
* There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called *pre-pruning*), or building the tree but then removing or collaps‐ ing nodes that contain little information (also called *post-pruning* or just *pruning*). scikit-learn only implements pre-pruning, not post-pruning.
* The most commonly used summary is *feature importance*, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target.”
* However, if a feature has a low value in feature\_importance\_, it doesn’t mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.
* from sklearn.tree import DecisionTreeClassifier, DecisionTreeClassifier(max\_depth=4,random\_state=0)
* from sklearn.tree import export\_graphviz
* export\_graphviz(tree,out\_file="tree.dot", class\_names=["malignant", "benign"], feature\_names=cancer.feature\_names, impurity=False, filled=True)
* import graphviz, with open("tree.dot") as f: dot\_graph=f.read(), display(graphviz.Source(dot\_graph))
* print(tree.feature\_importances\_)
* plt.barh(range(n\_features), model.feature\_importances\_, align='center')
* np.unique(y)

pages 81-90

* The DecisionTreeRegressor (and all other tree-based regression models) is not able to *extrapolate*, or make predictions outside of the range of the training data.
* The linear model approximates the data with a line, as we knew it would. The tree model, on the other hand, makes perfect predictions on the training data; we did not restrict the complexity of the tree, so it learned the whole dataset by heart. The tree has no ability to generate “new” responses, outside of what was seen in the training data.
* 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 forests: 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.
* 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.
* To build a tree, we first take what is called a *bootstrap sample* of our data. Instead of looking for the best test for each node, in each node the algorithm randomly selects a subset of the features, and it looks for the best possible test involving one of these features.
* To make a prediction using the random forest, the algorithm first makes a prediction for every tree in the forest. For regression, we can average these results to get our final prediction. For classification, a “soft voting” strategy is used. This means each algorithm makes a “soft” prediction, providing a probability for each possible output label. The probabilities predicted by all the trees are averaged, and the class with the highest probability is predicted.
* The feature importances provided by the random forest are more reliable than the ones provided by a single tree.
* Random forests don’t tend to perform well on very high dimensional, sparse data, such as text data.
* In general, it’s a good rule of thumb to use the default values: max\_features=sqrt(n\_features) for classification and max\_features=n\_features for regression.
* plt.semilogy(ram\_prices.date, ram\_prices.price)
* from sklearn.tree import DecisionTreeRegressor
* X\_train=data\_train.date[:,np.newaxis]
* from sklearn.ensemble import RandomForestClassifier
* X,y=make\_moons(n\_samples=100, noise=0.25, random\_state=3)
* forest=RandomForestClassifier(n\_estimators=5,random\_state=2)

pages 91-100

* The gradient boosted regression tree is another ensemble method that combines multiple decision trees to create a more powerful model. Despite the “regression” in the name, these models can be used for regression and classification.
* 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.
* They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly.
* learning\_rate, which controls how strongly each tree tries to correct the mistakes of the previous trees.
* A common approach is to first try random forests, which work quite robustly
* If you want to apply gradient boosting to a large-scale problem, it might be worth looking into the xgboost package and its Python interface
* Their main drawback is that they require careful tuning of the parameters and may take a long time to train.
* As with other tree-based models, it also often does not work well on high-dimensional sparse data.
* In contrast to random forests, where a higher n\_esti mators value is always better, increasing n\_estimators in gradient boosting leads to a more complex model, which may lead to overfitting.
* Adding nonlinear features to the representation of our data can make linear models much more powerful.
* from sklearn.ensemble import GradientBoostingClassifier
* gbrt=GradientBoostingClassifier(random\_state=0, learning\_rate=0.01)
* gbrt=GradientBoostingClassifier(random\_state=0, max\_depth=1)
* X\_new=np.hstack([X,X[:,1:]\*\*2])
* from mpl\_toolkits.mplot3d import Axes3D, axes3d -> figure=plt.figure() -> ax=Axes3D(figure, elev=-152, azim=-26) -> mask=y==0 -> ax.scatter(X\_new[mask,0], X\_new[mask,1], X\_new[mask,2], c='b', s=60, edgecolor='k') -> ax.scatter(X\_new[~mask,0], X\_new[~mask,1], X\_new[~mask,2], c='r',marker='^', s=60, edgecolor='k')