CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Machine Learning

Machine Learning

Lecture: Model Selection Continued

Ted Scully

Model Selection using Scikit Learn

- Using cross fold validation
- Hyper-parameter optimization
- **Nested Cross Fold Validation**
- **Using Pipelines**
- **Evaluation**

Hyper Parameter Optimization

- Hyper parameters (which are those parameters not directly learnt by the machine learning algorithm) can be determined through experimentation, in other words searching the possible values for the parameter in order to obtain the best cross-validation value.

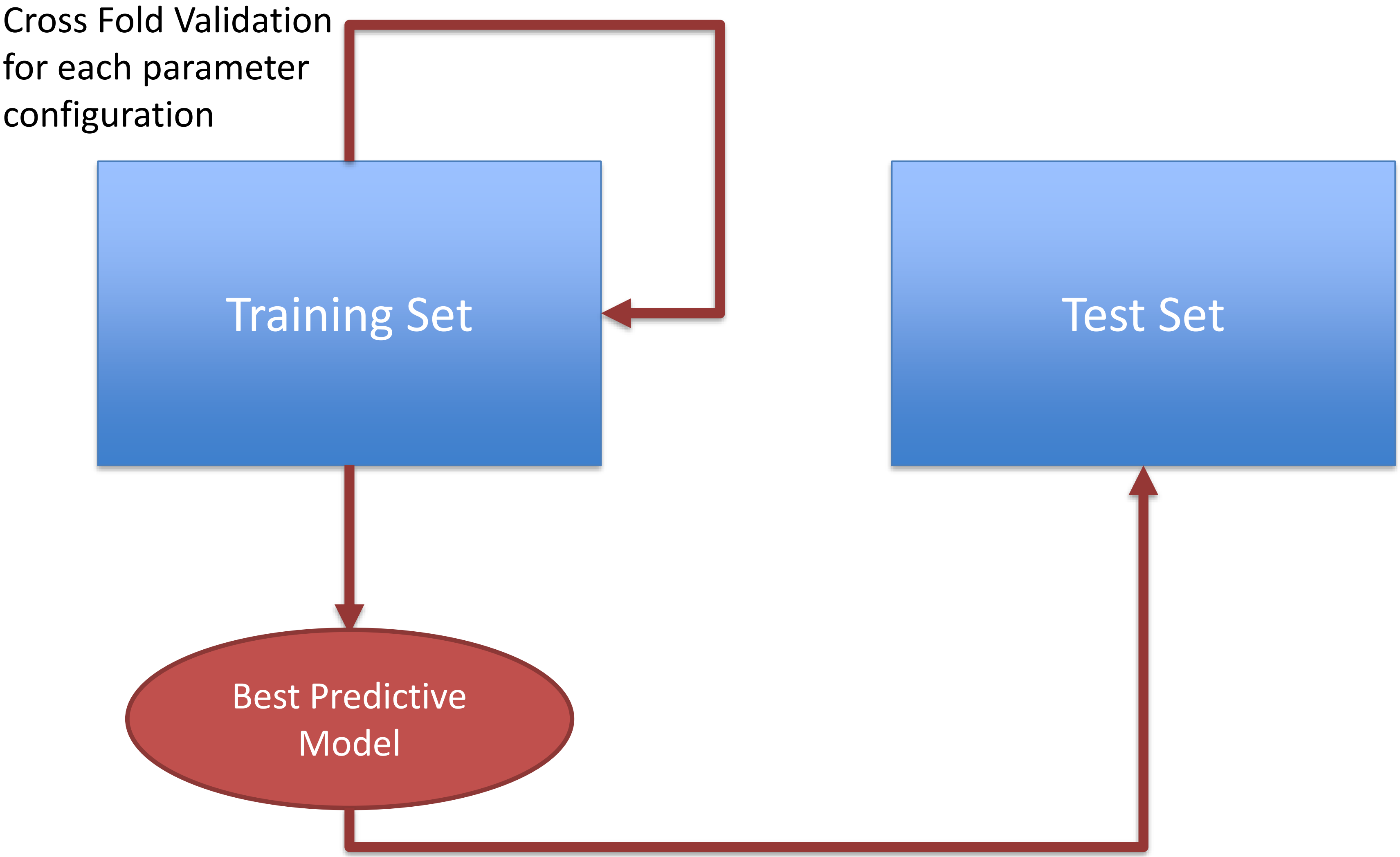
- Such independent parameters are often referred to as **hyper-parameters**.

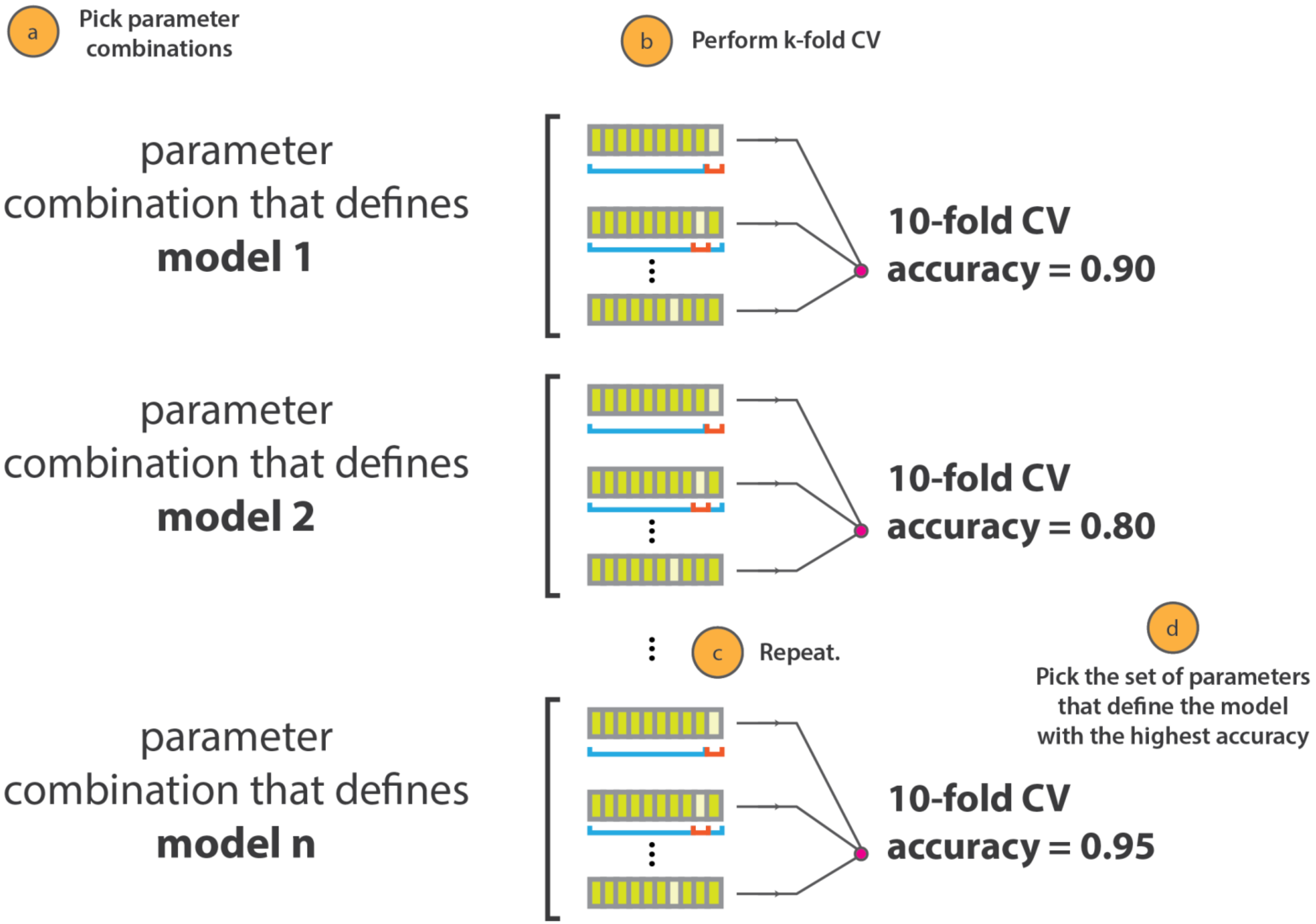
- A search process for the values of hyper-parameters consists of:
  - The learning algorithm
  - A parameter space (range of values for the parameter(s))
  - A method for searching these values
  - Cross Validation

- To find the names and current values for all parameters for a given estimator (ML Algorithm), use: **algorithm.get_params()** or the API pages.

- The learning algorithm
- A parameter space (range of values for the parameter(s))

High Level Hyper-Parameter Optimization with Cross Fold Validation





a Pick parameter combinations

b Perform k-fold CV

parameter combination that defines **model 1**

10-fold CV accuracy = 0.90

parameter combination that defines **model 2**

10-fold CV accuracy = 0.80

c Repeat.

d Pick the set of parameters that define the model with the highest accuracy

parameter combination that defines **model n**

10-fold CV accuracy = 0.95

https://cambridgecoding.wordpress.com/2016/04/03/scanning-hyperspace-how-to-tune-machine-learning-models/

Parameter Optimization

- The most commonly used method iof parameter optimization n Scikit-learn is **GridSearchCV (**sklearn.grid_search.GridSearchCV).

- It performs exhaustive search over a specified range of parameter values for an estimator.

- The grid search provided by GridSearchCV exhaustively generates candidates from a grid of parameter values specified with the **param_grid** parameter.
- The main input parameter to GridSearchCV is **param_grid** which is **list of dictionaries**

- Each dictionary has:
  - Parameters names as keys
  - Lists of parameter settings to try as values.
- You can include multiple grids.

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = model_selection.cross_val_score(knn, iris.data, iris.target, cv=10)
print (scores.mean())


param_grid = [ {'n_neighbors': list(range(1, 80)),  'p':[1, 2, 3, 4, 5] }  ]
clf = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10)

clf.fit(iris.data, iris.target)

print("\n Best parameters set found on development set:")
print(clf.best_params_ , "with a score of ", clf.best_score_)
```

```
from sklearn import model_selectio
from sklearn import datasets
from sklearn.neighbors import KNeig
from sklearn.model_selection impor

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = model_selection.cross_val_
print (scores.mean())


param_grid = [ {'n_neighbors': list(r
clf = GridSearchCV(KNeighborsClass

clf.fit(iris.data, iris.target)

print("\n Best parameters set found on development set:")
print(clf.best_params_ , "with a score of ", clf.best_score_)
```

```
0.966666666667

 Best parameters set found on development set
(note it is stored as a dictionary):
{'n_neighbors': 6, 'p': 3} with a score of  0.98
```

Note you can also use **clf. best_estimator_** and it will return the instance of the model that produced the best accuracy.

Note by default gridSearch will use the best parameters (identified using CV) and refit these to the entire dataset (this is what is return for clf.best_estimator_).

```
……….

iris = datasets.load_iris()

knn = KNeighborsClassifier()
scores = cross_validation.cross_val_score(knn, iris.data, iris.target, cv=10)
print scores.mean()



param_grid = [ {'n_neighbors': range(1, 80), 'p':[1, 2, 3, 4, 5]} ,
                                {'algorithm':['auto', 'ball_tree', 'kd_tree', 'brute'] } ]
```

Notice we can insert more than a single grid. In this example we have two separate grids. It is important to understand that each grid is searched separately

Parameter Optimization – Running jobs in parallel.

- **GridSearchCV** includes a parameter called **n_jobs**, which allows us to specify the number of jobs to run in **parallel**.
- By default the **n_jobs is set to 1**, which mean no joblib level parallelism is used at all.
- If set to -1, all available CPUs are used. Incorporating parallelism can significantly speed up your hyper-parameter optimization.
- For example, specifying the n_jobs=-1 below will provide a 3X speed-up on the computation search on my machine.

```
……….

param_grid = [ {'n_neighbors': list(range(1, 100)),  'p':[1, 2, 3, 4] }  ]

clf = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10, n_jobs=-1)

………
```

Model Selection using Scikit Learn

- Using cross fold validation
- Hyper-parameter optimization
- Nested Cross Fold Validation
- Using Pipelines
- Evaluation

Disadvantage of Cross Fold Validation

- Cross fold validation provides a less optimistic estimate of accuracy compared to just a simple train/test split or even a hold-out cross validation.

- However, it has still been shown that cross fold validation (by itself) can still positively bias the overall level of accuracy.

- One of the drawbacks of using cross fold validation is that it uses the same data to *tune* model parameters and evaluate model performance and as such can positively bias the accuracy estimation.

- The significance of this effect is primarily dependent on the size of the dataset and the stability of the model (see paper by Cawley and Talbot below).

On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation – Cawley and Talbot (2010)

Nested Cross Fold Validation

- To avoid this problem, we use nested cross fold validation which effectively uses a series of train/validation/test set splits.

- In nested cross-validation, we have an **outer** k-fold cross-validation loop to split the data into training and test folds.

- For each split we then have an **inner loop** that is used to perform grid search (perform hyper-parameter optimization) using k-fold cross-validation on the training data from the outer loop.

- After model selection, the test fold is then used to evaluate the model performance.

- The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyper-parameters of a model and then use it on unseen data.

Nested Cross Fold Validation

- Two important points about nested cross fold validation
  - It provides us with an unbiased estimation of accuracy (it does not provide us with an optimal model, this is still obtained using straight forward GridSearchCV).
  - It is computationally expensive.
    - Consider a grid search that has 300 possible combinations. With GridSearchCV we would end up building 300*10 (3000) models
    - Now consider if we use a nest cross fold validation with an outer number of loops of 10. With nested cross fold validation we would end up building 10*300*10 (30000) models.

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np

iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target


param_grid = [ {'n_neighbors': list(range(1, 80)),  'p':[1, 2, 3, 4, 5] }  ]

gridS = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10)

scores = model_selection.cross_val_score(gridS, X_iris, y_iris, cv=5)

print (scores.shape)

print (np.mean(scores))
```

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neighbors import KNeig
from sklearn.model_selection impor
import numpy as np

iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target


param_grid = [ {'n_neighbors': list(range(1, 80)),  'p':[1, 2, 3, 4, 5] }  ]

gridS = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10)

scores = model_selection.cross_val_score(gridS, X_iris, y_iris, cv=5)

print (scores.shape)

print (np.mean(scores))
```

Notice that we use an outer loop of 5 fold cross validation. For each iteration of this loop we perform **GridSearchCV**. Grid search will return the best model and that is then tested on the unseen data that has not been involved in the grid search process.
The overall accuracy is then averaged over the five folds.

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np

iris = datasets.load_iris()
X_iris = iris.data
y_iris = iris.target


param_grid = [ {'n_neighbors': list(range(1, 80)),  'p':[1, 2, 3, 4, 5] }  ]

gridS = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10)

scores = model_selection.cross_val_score(gridS, X_iris, y_iris, cv=5)

print (scores.shape)

print (np.mean(scores))
```

```
(5,)
0.973333333333
```

Model Selection using Scikit Learn

- Using cross fold validation
- Hyper-parameter optimization
- Nested Cross Fold Validation
- Using Pipelines
- Evaluation

Using Pipelines in Scikit Learn

- As we have seen we often have to perform various pre-processing techniques on a machine learning algorithm such as standardization, dimensionality reduction, encoding etc.
- Scikit contains a very useful tool in the form of the Pipeline class that facilitates this flow of operation.
- It allows us to fit a model including an arbitrary number of transformation/pre-processing steps and apply it to make predictions about new data.

- When creating a pipeline we pass it a list of tuples. Each tuple specifies:
  - A string identified that we can use to refer to the element of the pipeline
  - A transformer or estimator object

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score

X, y = load_breast_cancer(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
            ('clf', KNeighborsClassifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

```
from sklearn.model_selection im
from sklearn.preprocessing impo
from sklearn.decomposition imp
from sklearn.neighbors import K
from sklearn.pipeline import Pip
from sklearn.datasets import loa
from sklearn.metrics import accu

X, y = load_breast_cancer(return

X_train, X_test, y_train, y_test = train_tes          , y, test_size=0.20,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
            ('clf', KNeighborsClassifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

The `Pipeline` object takes a list of **tuples** as input, where the first value in each tuple is an arbitrary **identifier string** that we can use to access the individual elements in the pipeline. The second element in every tuple is a scikit-learn **transformer** or **estimator**.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import ac

X, y = load_breast_cancer(retu

X_train, X_test, y_train, y_test
random_state=1)

pipe_lr = Pipeline( [ ('scl', Star        caler()), ('pca', PCA(n_components=2)),
          ('clf', KNeighborsC      sifier()) ])

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

Fit all the transformations one after the other and transform the training data, then fit the transformed data using the final estimator (in other words build the ML model).

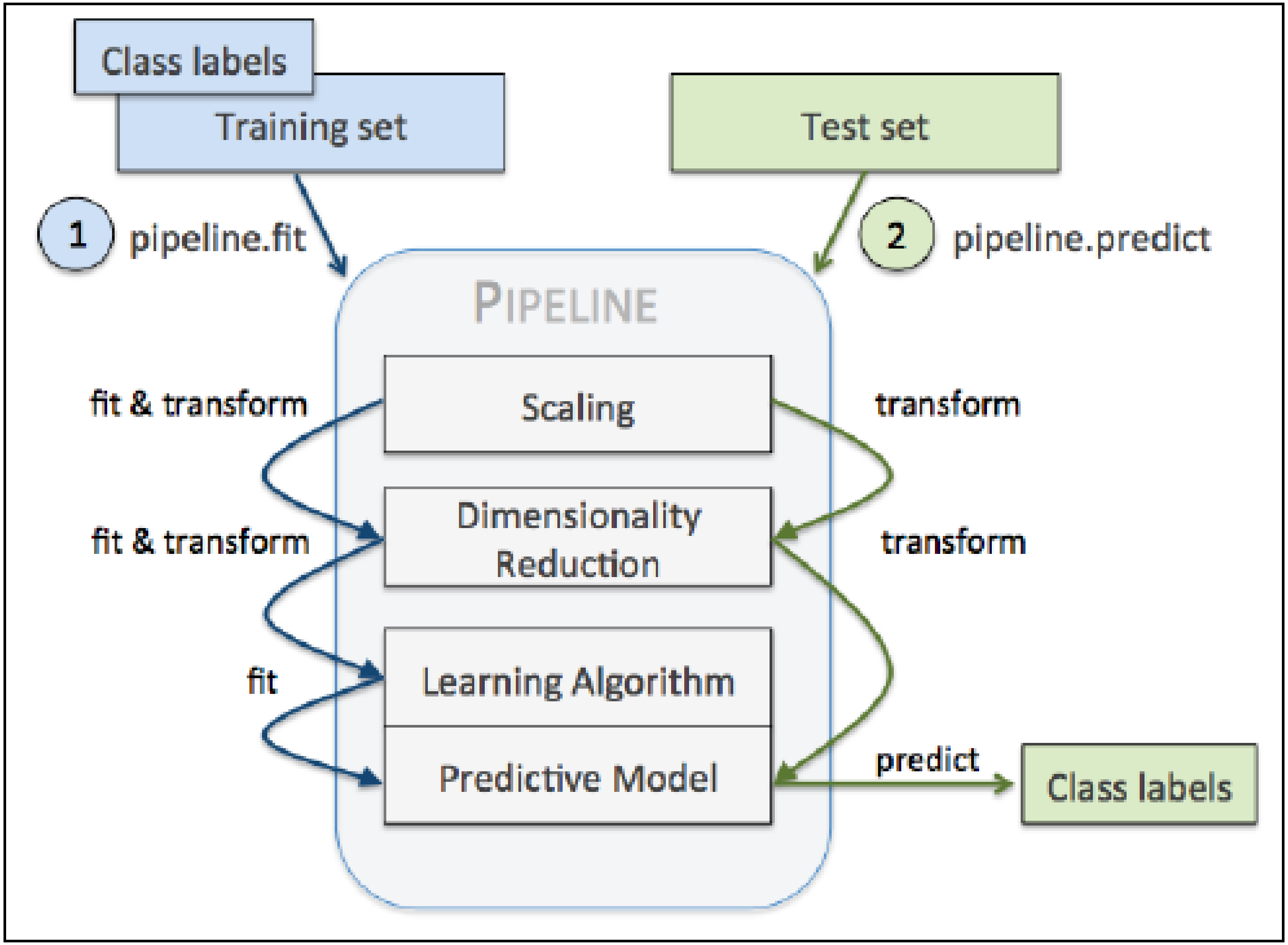
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors i
from sklearn.pipeline in
from sklearn.datasets ir
from sklearn.metrics im

X, y = load_breast_canc

X_train, X_test, y_train,
random_state=1)

pipe_lr = Pipeline( [ ('scl', StandardS          pca', PCA(n_components=2)),
          ('clf', KNeighborsClassifier())

pipe_lr.fit(X_train, y_train)
predictedResults = pipe_lr.predict(X_test)
print (accuracy_score(predictedResults, y_test))

# Alternatively we could substitute this line (instead of the last two lines)
print('Test Accuracy:',  pipe_lr.score(X_test, y_test))
```

Applies the transformations to the data, followed by the predict method of the final estimator in the pipeline. In other words the pipeline takes in the input data and transforms the data using each of the components in the pipeline. It then uses inputs this transformed data to the model built by the final estimator and it returns a set of predicted classes.

Using Pipelines

- The initial steps in a pipeline constitute scikit-learn **transformers**, and the last step is an **estimator**.
- In the example code, we built a pipeline that consisted of two intermediate steps, (i) a **StandardScaler** and (ii) a **PCA** transformer, and finally a **nearest neighbour** classifier as a final estimator.

- The following actions take place when we executed the fit method on the pipeline **pipe_lr**:
  - The StandardScaler performed fit and transform on the training data, and the transformed training data was then passed onto the next object in the pipeline, the PCA.
  - Similar to the previous step, PCA also executed fit and transform on the scaled input data and passed it to the final element of the pipeline, the estimator.
- There isn't any upper limit to the number of intermediate steps in this pipeline


Class labels
Training set
Test set
1
pipeline.fit
2
pipeline.predict
PIPELINE
fit & transform
Scaling
transform
fit & transform
Dimensionality Reduction
transform
fit
Learning Algorithm
Predictive Model
predict
Class labels

Using Pipelines for Cross Fold Validation

- (Notice in the previous example, we had test data to test the model produced from the pipeline)
- In many cases a pipeline is used to assemble several steps that can be cross-validated together while setting different parameters.

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score

X, y = load_breast_cancer(return_X_y=True)


pipe_lr = Pipeline([('scl', StandardScaler()), ('pca', PCA(n_components=2)),
        ('clf', KNeighborsClassifier())])


results = cross_val_score(pipe_lr, X, y, cv=10)

print(results.mean())
```

Using Pipelines with GridSearch

- We can also easily use a pipeline with grid search for hyper-parameter optimization.
- One of the benefits of this approach is that we can now incorporate parameters of the transformers in the search process.
- For this, a pipeline enables setting parameters of the various steps using their names and the parameter name separated by a '__', as in the example below.

```
from sklearn.model_selection import GridSearchCV

X, y = load_breast_cancer(return_X_y=True)


pipe_lr = Pipeline([('scl', StandardScaler()), ('pca', PCA(n_components=2)),
           ('clf', KNeighborsClassifier())])


param_grid = dict( pca__n_components=[2, 3, 4, 5], clf__n_neighbors= list(range(1, 30, 2)))

grid_search = GridSearchCV(pipe_lr, param_grid=param_grid)
grid_search.fit(X, y)
print(grid_search.best_estimator_, grid_search.best_score_)
```

Model Persistence

- After tuning the performance of your model you will most likely want to persist the model for future use.
- It is possible to save and load a scikitlearn model using **joblib** (joblib provide a replacement for pickle Python objects containing large data, in particular large numpy arrays).
- The following example show how we can save an svm model that we have tuned for the titanic dataset and then reload.

```
from sklearn.externals import joblib

param_grid = [ {'kernel': ['rbf', 'poly', 'linear'],  'C':range(1,15)}  ]
clf = GridSearchCV(SVC(), param_grid, cv=10)
clf.fit(data, target)


joblib.dump(clf.best_estimator_, 'titanic_svm.joblib')

loadedSVM = joblib.load('titanic_svm.joblib’)
```

Machine Learning

Machine Learning

Lecture: Evaluation

Ted Scully

Performance Evaluation Metrics

- Basic classification or misclassification accuracy hides quite a lot of detail.

- For example, an ML algorithm might be doing well are predicting one class but may be poor are predicting another class. This type of behaviour can often be masked when simply looking at classification accuracy.

- There are a range of different metrics that can be used to provide a more meaningful evaluation accuracy. Before we delve into these we will first introduce the notion of a confusion matrix.

Confusion Matrix

- Confusion matrix contains information about the actual and predicted classifications of a classification algorithm

- **True positives (TP)** - Number of instances of Category A that were correctly classified as Category A.
- **False Positives (FP) -** Number of instances of Category B that were incorrectly classified as Category A.
- **False Negative (FN)** is the number of instances of Category A that were incorrectly classified as Category B.
- **True Negative (TN)** is the number of instances of Category B that were correctly classified as Category B.

		Predicted	
		Category A	Category B
Actual	Category A	**TP**	**FN**
	Category B	**FP**	**TN**

Confusion Matrix Recap

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn import model_selection
from sklearn.metrics import confusion_matrix


X, y = load_breast_cancer(return_X_y=True)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
                                                           test_size=0.20, random_state=1)

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])
pipe_lr.fit(X_train, y_train)
y_predicted = pipe_lr.predict(X_test)

confmat = confusion_matrix(y_true=y_test, y_pred=y_predicted)
print (confmat)
```

The following code generates a confusion matrix:

```
[[37  5]
 [ 0 72]]
```

The array that was returned after executing the preceding code provides us with information about the different types of errors the classifier made on the dataset.

k Fold Cross Validation

- The code on the previous slide uses a holdout set. In most cases we will be interested in generating a confusion matrix for cross-fold validation.

- However, when using cross_val_score it only returns the individual accuracy for each fold, which is not sufficient for generating the confusion matrix.

- We can use a variant of cross_val_score, called **cross_val_predict** which returns the **prediction made for every single instance** as part of the cross fold validation. Remember with cross fold validation, each data instance is tested once.

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsClassifier

X, y = load_breast_cancer(return_X_y=True)
print ("Total instance in dataset is ",len(X))

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])

y_pred = cross_val_predict(pipe_lr, X, y, cv=10)

cnf_matrix = confusion_matrix(y, y_pred)

print (cnf_matrix)
```

```
[[197  15]
 [  4 353]]
```

Note the cross_val_predict score takes as input our pipeline object, the training data and labels and the number of folds. It then return an array containing the predicted value for each instance in the data test.

Accuracy Paradox

- Accuracy is a simplistic measure and should not be used in isolation.
- We have already seen this when dealing with highly imbalanced datasets.
- The accuracy of the model will appear high but the model is just predicting the majority class.

- The situation is often referred to as the accuracy paradox.

- It occurs when your algorithm reports a very high level of accuracy (such as 95%), but the accuracy is only reflecting the class distribution within the dataset.
- We have seen how we can identify this issue using a confusion matrix.

- We can use other metrics such as recall, precision and f value to gain **further insight** into the performance of a model.

Recall (Sensitivity or True Positive Rate)

Predicted

		Class A	Class B
Actual	Class A	True Positives	False Negatives
	Class B	False Positives	True Negatives

- Recall. Take all data instances belonging to a single class (Class A above for example). Recall tells us how many of these our model correctly predicted.
- Can be calculated as follows:

$$\frac{True\ Positives}{True\ Positives + False\ Negatives}$$

- The obvious advantage of this is that it will **highlight if we do very poorly on predicting any specific class**. For example, in the imbalanced dataset it would clearly show that our algorithm performs very badly on the minority class

Recall (Sensitivity or True Positive Rate)

		Predicted	
		Class A	Class B
Actual	Class A	True Positives	False Negatives
	Class B	False Positives	True Negatives

- Recall. Take all data instances belonging to a single class (Class A above for example). Recall tells us how many of these our mo
- Can be calculated as follows:

How confident we can be that all instances belonging to a specific class have been correctly classified by the model.

$$\frac{True\ Positives}{True\ Positives + False\ Negatives}$$

- The obvious advantage of this is that it will **highlight if we do very poorly on predicting any specific class**. For example, in the imbalanced dataset it would clearly show that our algorithm performs very badly on the minority class

Precision (Positive Predictive Value)

		Predicted	
		Class A	Class B
Actual	Class A	True Positives	False Negatives
	Class B	False Positives	True Negatives

- **Precision**. Take all data instances that our algorithm predicted were belong to a single class (Class A above for example). Precision tells us how many of these model correctly predicted.
- Can be calculated as follows:

$$\frac{True\ Positives}{True\ Positives + False\ Positives}$$

How confident we can be that any instance predicted as belonging to a certain class actually belongs to that class.

Predicted

		Spam	Ham
Actual	Spam	60	30
	Ham	20	90

- **Recall** measures how often the spam messages in the test set were actually marked as spam (60/(60+30)) = 0.667.
  - Recall can tell us how likely it is that a spam email will be missed by the system and end up in our inbox (1 - recall) = 0,333

- **Precision** measures how often the emails marked as spam are spam (60/(60+20)) = 0.75.
  - Precision tells us how likely it is that a genuine ham email could be marked as spam and potentially deleted (1-precision) = 0.25

F Score (F_1 Score)

- The F score takes recall and precision and gives you a single real number evaluation metric.

- Can be calculated as follows:

$$2 * \frac{(Recall)*(Precision)}{(Recall)+(Precision)}$$

Algorithm	Precision	Recall	F Score
A	0.4	0.6	0.48
B	0.7	0.2	0.31
C	0.01	1	0.0198

- For the F Score to be high both precision and recall need to be high.

Accessing Classification Metrics

- There are a number of ways we can access these classification metrics.

1. Directly through sklearn.metrics
2. Classification Report
3. Specifying a scoring parameter.

Using sklearn.metrics

- The sklearn.metrics module contains a large number of metrics that we can apply. Notice below we generate the f1_score.

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import f1_score


X, y = load_breast_cancer(return_X_y=True)
print ("Total instance in dataset is ",len(X))

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])

y_pred = cross_val_predict(pipe_lr, X, y, cv=10)

print (f1_score(y, y_pred))
```

```
Total instance in dataset is  569
0.973793103448
```

Using sklearn.metrics

- The sklearn.metrics module contains a large number of metrics that we can apply.

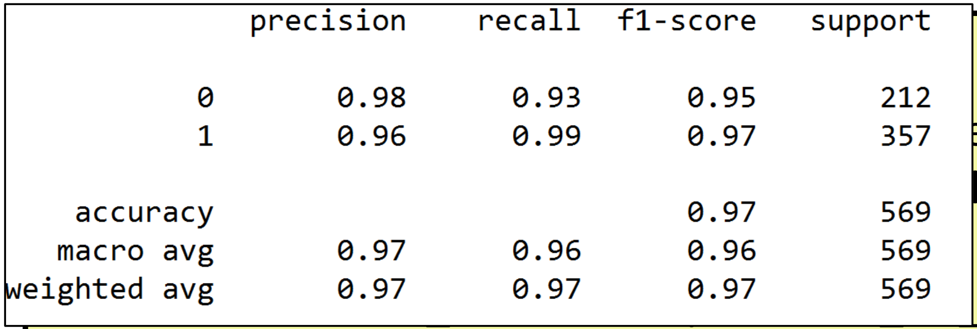
```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_pred
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report


X, y = load_breast_cancer(return_X_y=True)
print ("Total instance in dataset is ",len(X))

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])

y_pred = cross_val_predict(pipe_lr, X, y, cv=10)

print (classification_report(y, y_pred))
```

Another useful method that you can apply in exactly the same way as the f1_score here is **sklearn.metrics.classification_report,** which provides and overview of f1, recall and precision broken down by class.

Using sklearn.metrics

- The sklearn.metrics module contains a large number of metrics that we can apply.

Another useful method that you can apply in exactly the same way as the f1_score here is **sklearn.metrics.classification_report,** which provides and overview of f1, recall and precision broken down by class.



```
              precision    recall  f1-score   support

           0       0.98      0.93      0.95       212
           1       0.96      0.99      0.97       357

    accuracy                           0.97       569
   macro avg       0.97      0.96      0.96       569
weighted avg       0.97      0.97      0.97       569
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report



X, y = load_breast_cancer(return_X_y=True)
print ("Total instance in dataset is ",len(X))

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])

y_pred = cross_val_predict(pipe_lr, X, y, cv=10)

print (classification_report(y, y_pred))
```

Directly specifying scoring function

Cross-fold validation will automatically use the default accuracy metric of the estimator object but you can also set it to use a specific scoring function. See below we use precision, recall and f1.

```
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

X, y = load_breast_cancer(return_X_y=True)

pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])

precision_scores = cross_val_score(pipe_lr, X, y, cv=10, scoring='precision')
recall_scores = cross_val_score(pipe_lr, X, y, cv=10, scoring='recall')
f_scores = cross_val_score(pipe_lr, X, y, cv=10, scoring='f1')

print ("Average precision score ", precision_scores.mean())
print ("Average recall score ", recall_scores.mean())
print ("Average F1 score ", f_scores.mean())
```

```
Average precision score  0.96
Average recall score  0.988
Average F1 score  0.97
```

It is also useful to note that we can directly specify the scoring metric we might use for GridSearchCV as well.

By default it is just going to use accuracy. However, we can directly specify other metrics such as f1 below

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

X, y = load_breast_cancer(return_X_y=True)
pipe_lr = Pipeline([('scl', StandardScaler()), ('clf', KNeighborsClassifier())])
param_grid = [ {'clf__n_neighbors': list(range(1, 5)),  'clf__p':[1, 2, 3, 4, 5] }  ]
clf = GridSearchCV(pipe_lr, param_grid, cv=10, scoring= 'f1')
clf.fit(X, y)

print("\n Best parameters set found on development set:")
print(clf.best_params_ , "with a score of ", clf.best_score_)
```

It is also useful to note that we can directly specify the scoring metric we might use for GridSearchCV as well.

By default it is just going to use accuracy. However, we can directly specify other metrics such as f1 below

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier

X, y = load_breast_cancer(return_X_y=True)
pipe_lr = Pipeline([('scl', StandardScaler()), ('c
param_grid = [ {'clf__n_neighbors': list(range
clf = GridSearchCV(pipe_lr, param_grid, cv=1
clf.fit(X, y)

print("\n Best parameters set found on devel
print(clf.best_params_ , "with a score of ", clf.
```

To get a full list of the scoring metrics that you can use in Cross Fold or Grid Search you can use the following:

```
from sklearn.metrics import SCORERS
print (SCORERS.keys())
```

Basic Measures of Regression

- A basic performance measure used for regression is the **mean squared error (MSE).** It allows us to rank the performance of multiple models on a prediction problem with a continuous value.

$$\frac{1}{m}\sum_{i=0}^{m}((f\left(x^{i}\right) - y^{i}))^{2}$$

Basic Measures of Regression

- A basic performance measure used for regression is the **mean squared error (MSE).** It allows us to rank the performance of multiple models on a prediction problem with a continuous value.

$$\frac{1}{m}\sum_{i=0}^{m}((f\left(x^{i}\right) - y^{i}))^{2}$$

- Although this measures allows us to rank different models, one **drawback is that the error values themselves are not particularly meaningful** in relation to the scenario the model is being used for.

- For example, if we are trying to **predict rental prices for office space**, the MSE value cannot give us an estimation of the number of euros by which the model is incorrect in it’s predictions (The reason for this is due to the squared term in the mean squared error calculation).

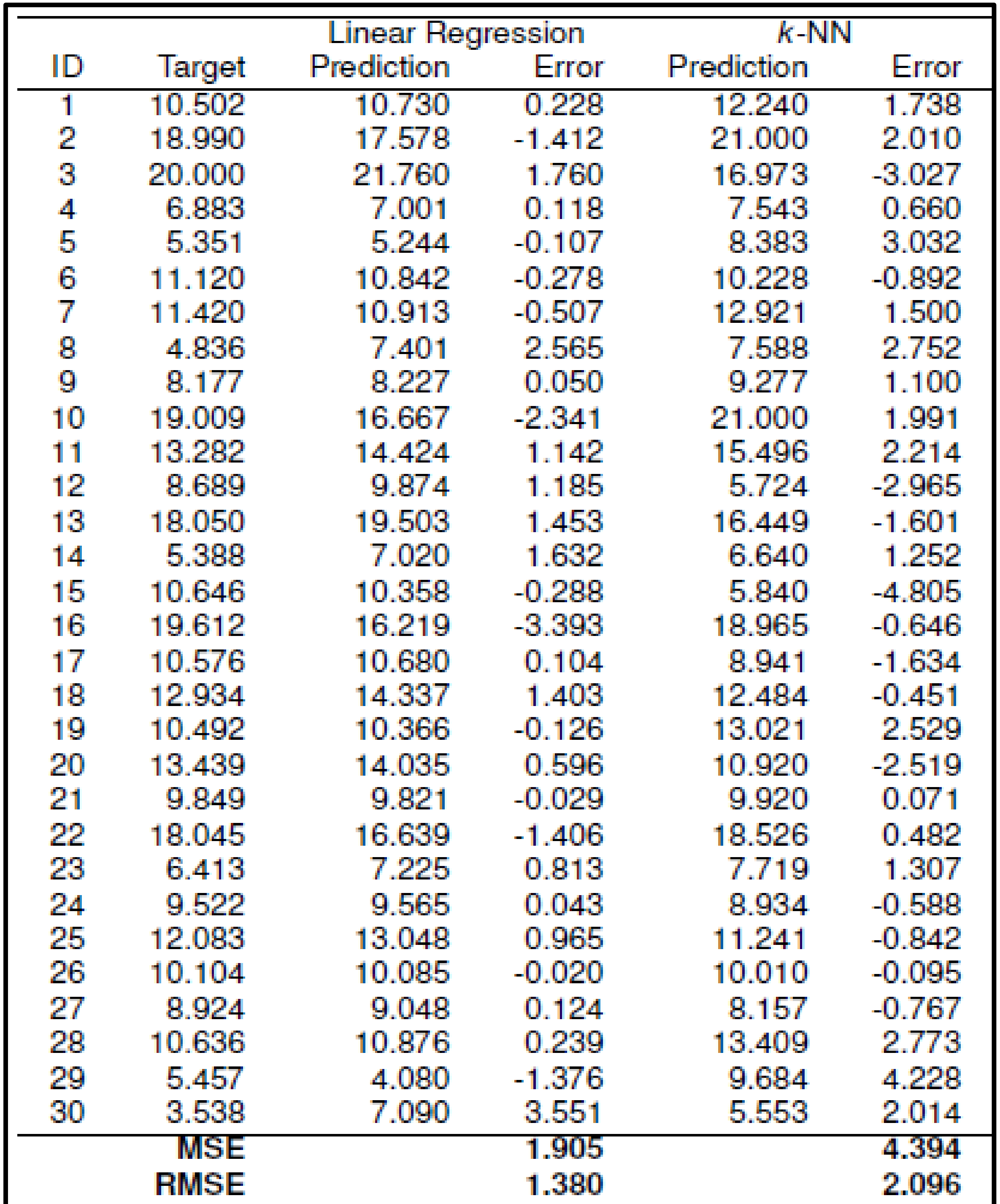
Basic Measures of error

- The problem outlined in the previous slide can be rectified by using the root mean squared error instead. The root mean squared error (**RMSE**) is calculated as:

$$\sqrt{\frac{1}{m}\sum_{i=0}^{m}((f(x^{i}) - y^{i}))^{2}}$$

- RMSE values are in the **<u>same units as the target value</u>** and so allow us to say something more meaningful about the errors being made by the model.
- It should be noted that the RMSE tends to overestimate error slightly as it tends to overemphasise large errors (due to the squared value).

The example shown on the right shows the expected target values for a test set, the predictions made by two different models (one a linear regression and the other a k-NN). The prediction problem in this case is to determine **the dosage of a blood thinning drug (in milligrams)** that should be given to a patient.

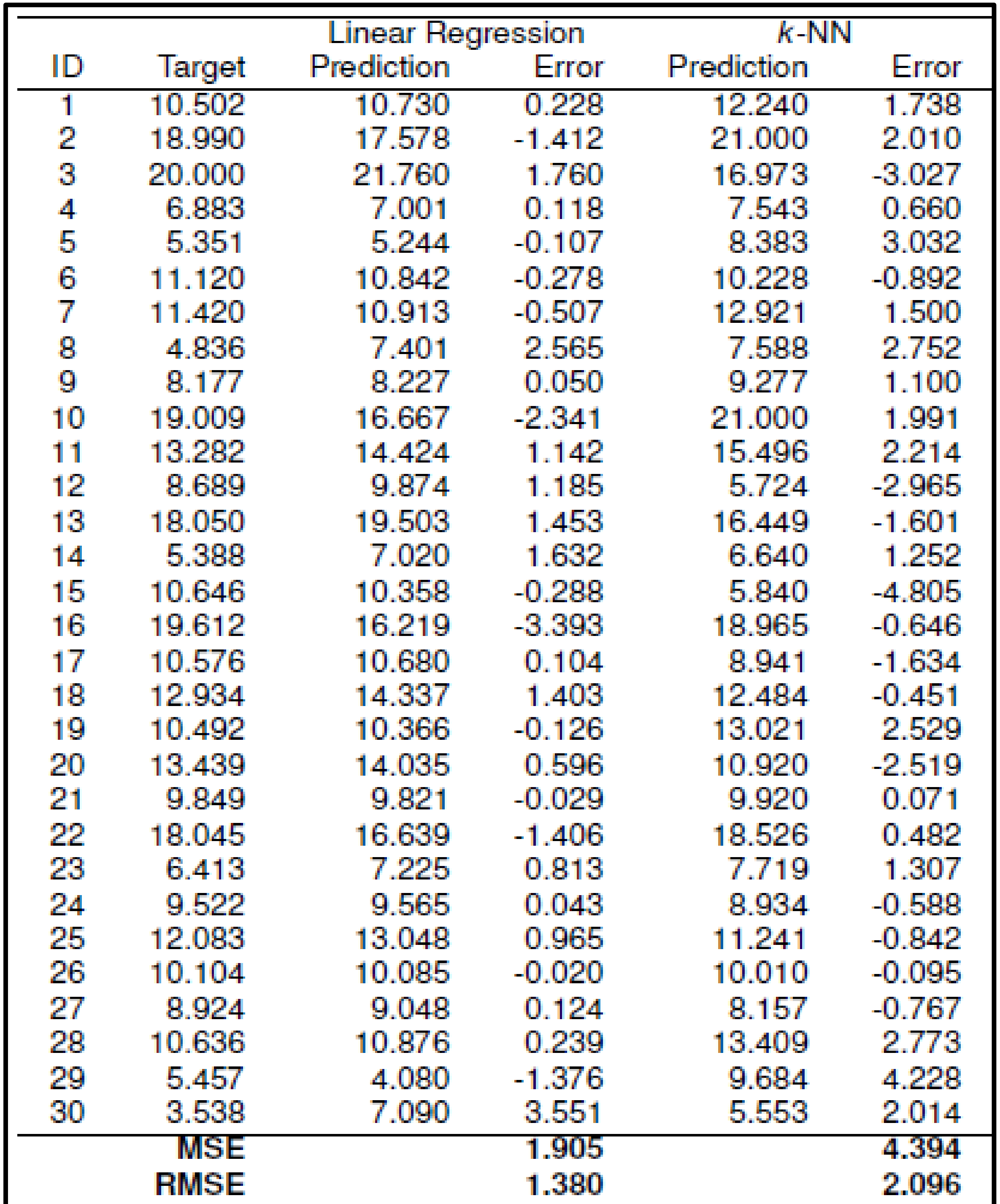


		Linear Regression		*k*-NN	
ID	Target	Prediction	Error	Prediction	Error
1	10.502	10.730	0.228	12.240	1.738
2	18.990	17.578	-1.412	21.000	2.010
3	20.000	21.760	1.760	16.973	-3.027
4	6.883	7.001	0.118	7.543	0.660
5	5.351	5.244	-0.107	8.383	3.032
6	11.120	10.842	-0.278	10.228	-0.892
7	11.420	10.913	-0.507	12.921	1.500
8	4.836	7.401	2.565	7.588	2.752
9	8.177	8.227	0.050	9.277	1.100
10	19.009	16.667	-2.341	21.000	1.991
11	13.282	14.424	1.142	15.496	2.214
12	8.689	9.874	1.185	5.724	-2.965
13	18.050	19.503	1.453	16.449	-1.601
14	5.388	7.020	1.632	6.640	1.252
15	10.646	10.358	-0.288	5.840	-4.805
16	19.612	16.219	-3.393	18.965	-0.646
17	10.576	10.680	0.104	8.941	-1.634
18	12.934	14.337	1.403	12.484	-0.451
19	10.492	10.366	-0.126	13.021	2.529
20	13.439	14.035	0.596	10.920	-2.519
21	9.849	9.821	-0.029	9.920	0.071
22	18.045	16.639	-1.406	18.526	0.482
23	6.413	7.225	0.813	7.719	1.307
24	9.522	9.565	0.043	8.934	-0.588
25	12.083	13.048	0.965	11.241	-0.842
26	10.104	10.085	-0.020	10.010	-0.095
27	8.924	9.048	0.124	8.157	-0.767
28	10.636	10.876	0.239	13.409	2.773
29	5.457	4.080	-1.376	9.684	4.228
30	3.538	7.090	3.551	5.553	2.014
	MSE		**1.905**		**4.394**
	RMSE		**1.380**		**2.096**

Notice we can see from both the MSE and RMSE that the LR model performs better than the k-NN model.

The useful aspect about the RMSE is that we can derive additional meaning from the results. The predictions showed the **LR model to be 1.38 mg out on average**, whereas those made by the k-NN will be 2.096 mg out on average.

RMSE values are in the **same units as the target value**



		Linear Regression		*k*-NN	
ID	Target	Prediction	Error	Prediction	Error
1	10.502	10.730	0.228	12.240	1.738
2	18.990	17.578	-1.412	21.000	2.010
3	20.000	21.760	1.760	16.973	-3.027
4	6.883	7.001	0.118	7.543	0.660
5	5.351	5.244	-0.107	8.383	3.032
6	11.120	10.842	-0.278	10.228	-0.892
7	11.420	10.913	-0.507	12.921	1.500
8	4.836	7.401	2.565	7.588	2.752
9	8.177	8.227	0.050	9.277	1.100
10	19.009	16.667	-2.341	21.000	1.991
11	13.282	14.424	1.142	15.496	2.214
12	8.689	9.874	1.185	5.724	-2.965
13	18.050	19.503	1.453	16.449	-1.601
14	5.388	7.020	1.632	6.640	1.252
15	10.646	10.358	-0.288	5.840	-4.805
16	19.612	16.219	-3.393	18.965	-0.646
17	10.576	10.680	0.104	8.941	-1.634
18	12.934	14.337	1.403	12.484	-0.451
19	10.492	10.366	-0.126	13.021	2.529
20	13.439	14.035	0.596	10.920	-2.519
21	9.849	9.821	-0.029	9.920	0.071
22	18.045	16.639	-1.406	18.526	0.482
23	6.413	7.225	0.813	7.719	1.307
24	9.522	9.565	0.043	8.934	-0.588
25	12.083	13.048	0.965	11.241	-0.842
26	10.104	10.085	-0.020	10.010	-0.095
27	8.924	9.048	0.124	8.157	-0.767
28	10.636	10.876	0.239	13.409	2.773
29	5.457	4.080	-1.376	9.684	4.228
30	3.538	7.090	3.551	5.553	2.014
	MSE		**1.905**		**4.394**
	RMSE		**1.380**		**2.096**

Basic Measures of error

- An alternative to the RMSE is called the mean absolute error (MAE) which can be calculated as:

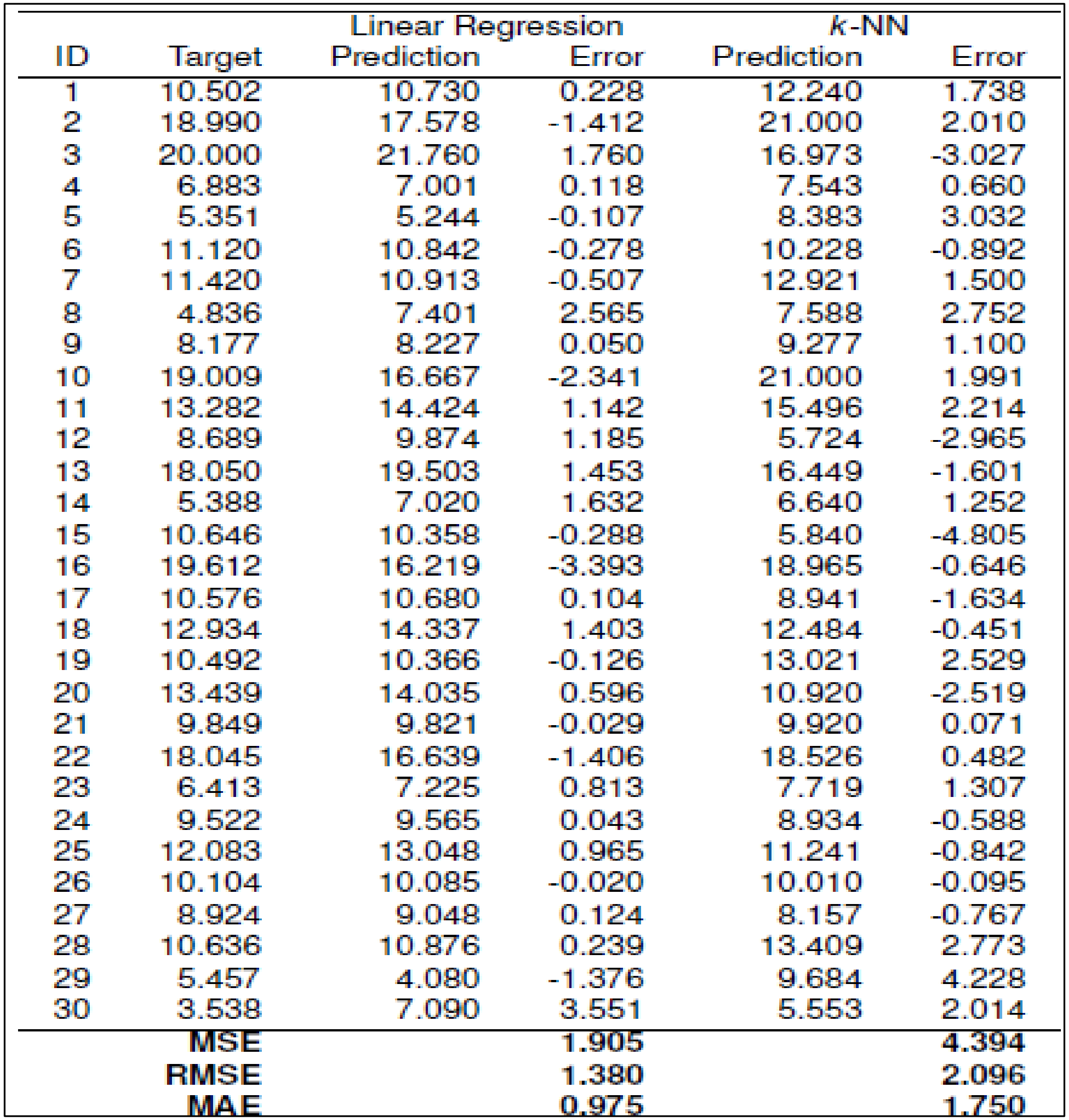
$$\frac{1}{m}\sum_{i=0}^{m}\left|f\left(x^{i}\right)-y^{i}\right|$$

- Interestingly the RMSE tends to be used more often than the MAE.

- One of the reasons for this is that it is considered better practice to be **pessimistic** about the performance of a model.

- Also RMSE has the benefit of **penalizing large errors** more than MAE.

Notice the MAE error for LR and kNN is less than the RMSE. The RMSE tends to have a more pessimistic view of error.



		Linear Regression		*k*-NN	
ID	Target	Prediction	Error	Prediction	Error
1	10.502	10.730	0.228	12.240	1.738
2	18.990	17.578	-1.412	21.000	2.010
3	20.000	21.760	1.760	16.973	-3.027
4	6.883	7.001	0.118	7.543	0.660
5	5.351	5.244	-0.107	8.383	3.032
6	11.120	10.842	-0.278	10.228	-0.892
7	11.420	10.913	-0.507	12.921	1.500
8	4.836	7.401	2.565	7.588	2.752
9	8.177	8.227	0.050	9.277	1.100
10	19.009	16.667	-2.341	21.000	1.991
11	13.282	14.424	1.142	15.496	2.214
12	8.689	9.874	1.185	5.724	-2.965
13	18.050	19.503	1.453	16.449	-1.601
14	5.388	7.020	1.632	6.640	1.252
15	10.646	10.358	-0.288	5.840	-4.805
16	19.612	16.219	-3.393	18.965	-0.646
17	10.576	10.680	0.104	8.941	-1.634
18	12.934	14.337	1.403	12.484	-0.451
19	10.492	10.366	-0.126	13.021	2.529
20	13.439	14.035	0.596	10.920	-2.519
21	9.849	9.821	-0.029	9.920	0.071
22	18.045	16.639	-1.406	18.526	0.482
23	6.413	7.225	0.813	7.719	1.307
24	9.522	9.565	0.043	8.934	-0.588
25	12.083	13.048	0.965	11.241	-0.842
26	10.104	10.085	-0.020	10.010	-0.095
27	8.924	9.048	0.124	8.157	-0.767
28	10.636	10.876	0.239	13.409	2.773
29	5.457	4.080	-1.376	9.684	4.228
30	3.538	7.090	3.551	5.553	2.014
	MSE		**1.905**		**4.394**
	RMSE		**1.380**		**2.096**
	MAE		**0.975**		**1.750**

Basic Measures of error

- An advantage of using **MAE and RMSE** is that the error values are in the **same units as the domain** (in the example on the previous slide they are in units of milligrams).

- The disadvantage of each of the measures described so far is that it is very difficult to know if the models are providing accurate predictions without a **knowledge of the domain**.

- For example, how do we know if the LR model on the previous slide is actually making accurate prediction without understanding the drug dosage domain.

- The **R^2** coefficient is a **domain independent measure of model performance** that is frequently used for prediction problems with a continuous target.

Basic Measures of error

- The **R²** coefficient compares the **performance of a model on a test set (sum of squared residuals)** with the performance of an imaginary model that always predicts the **average values from the test set (total sum of squares)**.

- The **R²** coefficient is calculated as:

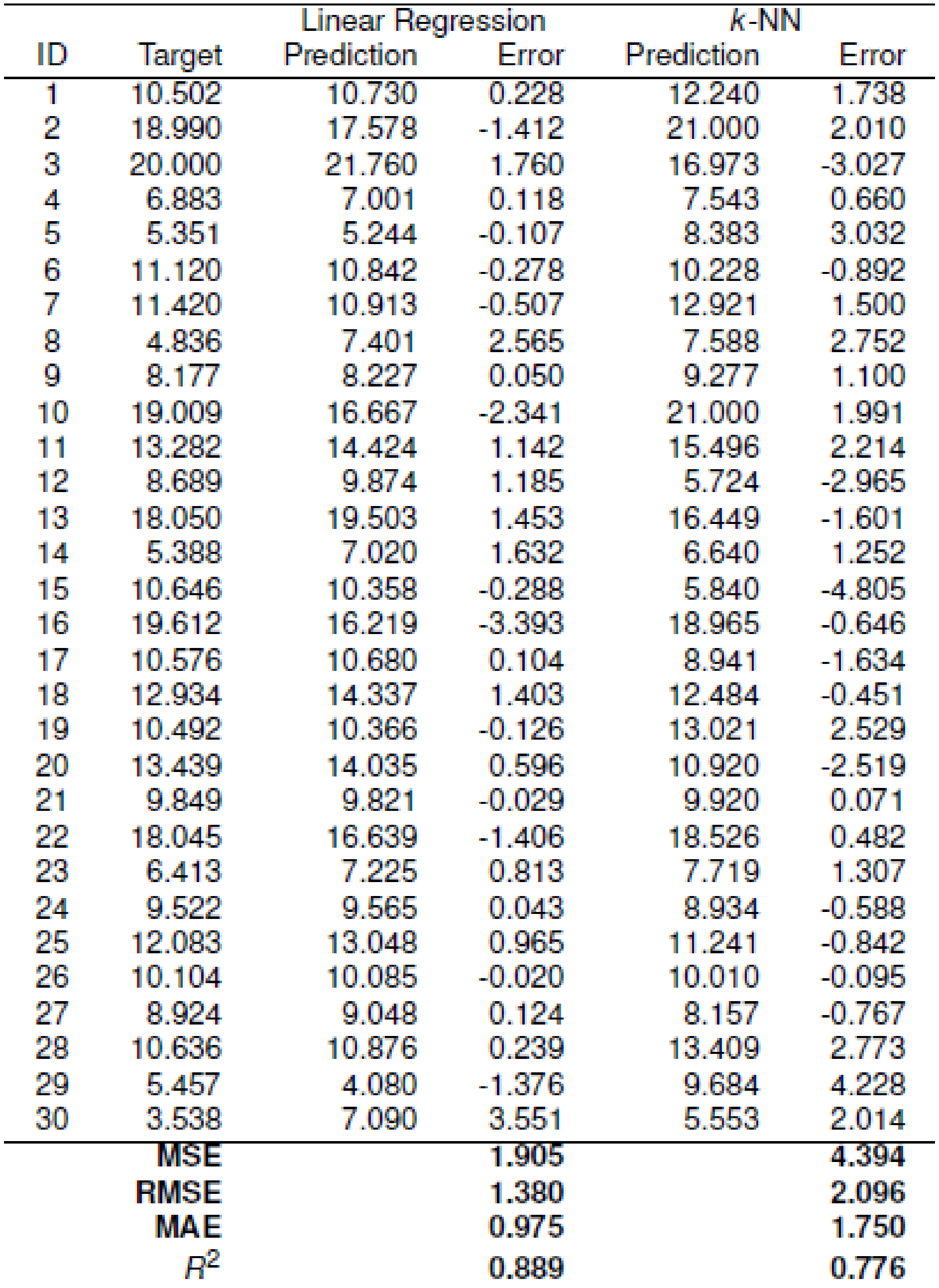
$$R^2 = 1 - \frac{sum\ of\ squared\ residuals}{total\ sum\ of\ squares}$$

- Where

$$sum\ of\ squared\ residuals = \sum_{i=0}^{m} (f\left(x^i\right) - y^i)^2$$

$$total\ sum\ of\ squares = \sum_{i=0}^{m} (\bar{y} - y^i)^2$$

The R2 result again indicates that the Linear Regression model outperforms the k-NN algorithm.



		Linear Regression		*k*-NN	
ID	Target	Prediction	Error	Prediction	Error
1	10.502	10.730	0.228	12.240	1.738
2	18.990	17.578	-1.412	21.000	2.010
3	20.000	21.760	1.760	16.973	-3.027
4	6.883	7.001	0.118	7.543	0.660
5	5.351	5.244	-0.107	8.383	3.032
6	11.120	10.842	-0.278	10.228	-0.892
7	11.420	10.913	-0.507	12.921	1.500
8	4.836	7.401	2.565	7.588	2.752
9	8.177	8.227	0.050	9.277	1.100
10	19.009	16.667	-2.341	21.000	1.991
11	13.282	14.424	1.142	15.496	2.214
12	8.689	9.874	1.185	5.724	-2.965
13	18.050	19.503	1.453	16.449	-1.601
14	5.388	7.020	1.632	6.640	1.252
15	10.646	10.358	-0.288	5.840	-4.805
16	19.612	16.219	-3.393	18.965	-0.646
17	10.576	10.680	0.104	8.941	-1.634
18	12.934	14.337	1.403	12.484	-0.451
19	10.492	10.366	-0.126	13.021	2.529
20	13.439	14.035	0.596	10.920	-2.519
21	9.849	9.821	-0.029	9.920	0.071
22	18.045	16.639	-1.406	18.526	0.482
23	6.413	7.225	0.813	7.719	1.307
24	9.522	9.565	0.043	8.934	-0.588
25	12.083	13.048	0.965	11.241	-0.842
26	10.104	10.085	-0.020	10.010	-0.095
27	8.924	9.048	0.124	8.157	-0.767
28	10.636	10.876	0.239	13.409	2.773
29	5.457	4.080	-1.376	9.684	4.228
30	3.538	7.090	3.551	5.553	2.014
	MSE		**1.905**		**4.394**
	RMSE		**1.380**		**2.096**
	MAE		**0.975**		**1.750**
	R^2		**0.889**		**0.776**

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets.samples_generator import make_regression
from sklearn.metrics import r2_score
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split

data, target = make_regression(n_samples=10000, n_features=20, noise = 0.2,
random_state = 10)

train_features, test_features, train_labels, test_labels = train_test_split( data, target,
test_size=0.2)

reg = SVR()
reg = reg.fit(train_features, train_labels)

results= reg.predict(test_features)
print (r2_score(test_labels,results))
```

In this example we directly generate the R^2 value for the test set. We pass the r2_score the test regression values and the predicted results.

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets.samples_generator import make_regression
from sklearn.svm import SVR
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score



X, y = make_regression(n_samples=10000, n_features=20, noise = 0.2, random_state = 10)

rg = SVR()
y_pred = cross_val_predict(rg, X, y, cv=10)

print (mean_absolute_error(y, y_pred))
print (r2_score(y, y_pred))
```

This code is similar to the code we used in the classification slides. The only difference is that we are using the R^2 metric from the Scikit-Learn metrics package.

```
57.1673614688
0.783351369722
```

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets.samples_generator import make_regression
from sklearn.svm import SVR
import numpy as np


X, y = make_regression(n_samples=10000, n_features=20, noise = 0.2, random_state = 10)

clf = SVR()
scores = cross_val_score(pipe_lr, X, y, cv=10, scoring='r2')

print (np.mean(scores))
```

In this code we specify directly the scoring function we want to use in the cross validation. (We can perform the same with GridSearchCV)

0.783107141061