Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow
<u>Reading Notes</u>

Chapter 1: The Machine Learning Landscape

What is Machine Learning?

- Programming computers to learn from data

The data that a system uses to train is called the training set.

Applying machine learning techniques to dig into large amounts of data can help discover patterns that were not immediately apparent - This is called **data mining**.

Machine Learning is great for:

- Problems that require lots of tuning or lots of rules
- Complex problems for which traditional approaches don't work well
- Fluctuating environments
- Gaining insights about complex problems and large amounts of data

<u>Types of Machine Learning</u>

Supervised Learning

The training data you feed to the algorithm includes the desired solutions called labels.

Typical tasks:

- Classification (target a **class**)
- Target a **value** from 'features / predictors"
    - This type of task is called 'Regression'
    - Requires lots of data and labels

Some regression algorithms can also be used for classification.

Most important supervised learning algorithms:

- K-Nearest Neighbor
- Linear Regression

- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees & Random Forests
- Neural Networks

Unsupervised Learning

Training data is not labeled

Most important unsupervised learning algorithms:
- Clustering
    - K-Means
    - DBSCAN
    - Hierarchical Cluster Analysis (HCA)
- Anomaly detection & novelty detection
    - One-class SVM
    - Isolation forest
- Visualization and Dimensionality Reduction
    - Principal component analysis (PCA)
    - Kernel PCA
    - Locally - linear embedding (LLE)
    - T-distributed stochastic neighbor embedding (t-SNE)
- Association rule learning
    - Apriori
    - Eclat

A related task is **dimensionality reduction** in which the goal is to simplify the data without losing too much information. This can be done by merging correlated features / attributes into one. This is called **feature extraction** and is done by **dimensionality reduction algorithms**.

It is a good idea to reduce the dimensionality of your training data before feeding it to another machine learning algorithm.

Another important unsupervised task is **anomaly detection**
- Used to detect defects or fraud
- Also good to remove outliers from a training set before feeding it to another machine learning algorithm

Another important & similar unsupervised task is **novelty detection**.

- Used to detect new instances
- Requires a very ‘clean’ training set

Another important unsupervised task is **association rule learning** used to dig into large amounts of data to discover interesting relations between attributes.

Semi Supervised Learning

Partially labeled training data:

- Lots of unlabeled data
- Little bit of labeled data

Ex. Google photos

Most **semi supervised** learning algorithms are combinations of unsupervised and supervised algorithms.

Ex. Deep Belief Networks (DBNs) are made of unsupervised components called restricted boltzmann machines (RBMs) stacked on top of each other - they are trained sequentially and unsupervised, then, they are combined and fine tuned using supervised learning algorithms.

Reinforcement Learning

Reinforcement learning involves an agent that can observe the environment, select and perform actions, and get rewards in return (or penalties)

It must learn the best strategy called a **policy**

Ex. Learning how to walk or play Go

Steps:

- Observe
- Select action using policy
- Perform action
- Get reward or penalty
- Update policy (learning step)
- Iterate until optimal policy is found

Batch and Online Learning

Batch Learning: Does not learn incrementally, must be trained using all available data and retrained on old and new data (also called offline learning)

Online Learning: Train the system incrementally by feeding it data sequentially in small groups called mini batches

- Great for systems that receive a continuous flow of data that needs to change rapidly or autonomously
- Also good for situations in which a huge data set cannot fit in a single machines main memory (Out of core learning)
- After the training set is done the data is discarded
- Really called Incremental Learning
- How fast a system should adapt to changing data is called the **learning rate**
    - You need to watch out for bad data and adjust the learning rate to prevent a decline in performance
    - It is a good idea to preprocess data with an anomaly detection algorithm

Instance-Based Learning

- The system learns examples 'by heart' and then generalizes to new cases by how similar they are (using a measure of similarity)

Model-Based Learning

- The system builds a model from examples and then uses the model to make new predictions

Main Challenges of Machine Learning

- Bad learning algorithm
- Bad data

Bad data:

- Insufficient quantity of training data
- Nonrepresentative training data
    - Training data must be representative of the cases you want to generalize to.
        - If the sample is too small, you will have **sampling noise** (nonrepresentative data as a result of chance)
        - **Sampling bias** comes from large sets of data that are not representative because the sampling method is flawed.

- Pool quality data
    - Poor collection or uncleaned data
- Irrelevant features give you garbage out
    - Need to do **Feature Engineering**
        - Feature selection: Selecting the most useful features
        - Feature extraction: combining existing features to make new ones
        - Creating new features by gathering new data

Bad algorithms:

- **Overfitting** the training data
    - Overgeneralizing
    - Happens when the model is too complex relative to the amount and noise of the training data.
        - **Regularization** reduces the risk of overfitting
            - The amount of regularization to apply during learning can be controlled by a **hyperparameter**
            - A **hyperparameter** is a parameter of a learning algorithm (not the model itself)
- **Underfitting** the training data
    - Happens when your model is too simple to learn the underlying structure of the data

Testing and Validating

Break your dataset into training and testing data

The error rate on new cases is called the **generalization error**

- If the training error is low (i.e. your model makes few mistakes on the training set) but the generalization error is high, it means your model is overfitting the training data.

A common practice is to use 80% of your data for training and 20% for testing, but it depends on the size of the dataset.

Hyperparameter tuning and model selection

How do you pick the right model?

- You need to try a bunch and make assumptions

- Holdout validation / validation set: Hold out part of the training set to evaluate candidate models

Chapter 2: End to End Machine Learning Project

Task: To use California census data to build a model of housing prices in the state.

Pipelines are a sequence of data processing components.

This task is a:

- Supervised learning task
- Regression task
    - Multiple regression problem (uses many features to make a prediction)
    - Also univariate regression problem (only trying to predict a single value)
        - (multivariate -> predict multiple values)
- Batch learning

Select a performance measure

A typical performance measure is Root Mean Square Error (RMSE)

RMSE(X,h) = $\sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)})-y^{(i)})^{2}}$

where

m: number of instances in the dataset

x^i: is a vector of all feature values of the ith instance

y^i: is the instance label

X: is a matrix containing all the feature values (excluding labels) of all instances in the dataset

h: is your system’s prediction function also called the **hypothesis**

RMSE(X,h): is the **cost function** measured on the set of examples using your hypothesis h.

Another good performance measure for regression tasks is Mean Absolute Error:

MAE(X,h) = $\frac{1}{m}\sum_{i=1}^{m}\left|h(x^{(i)})-y^{(i)}\right|$

Check your assumptions

List and verify the assumptions that have been made so far by you and others to catch serious issues early on.

Create a test set

We split the data set into a training set and test set (There are various methods to do this)

If you use purely random sampling methods, you need to watch out for **sampling bias** in the test set.

You should use **stratified sampling** to divide the instances into **strata** which are representative of the ratios of the overall dataset. (This also requires the larger dataset to have **strata** that are accurate)

Don't neglect making a good test set!

Typically you use 20% of your data for the test set.

Discover and Visualize the Data to gain Insights (See the Jupyter notebook)

Look for correlations

Compute the **standard correlation coefficient** (also called Pearson's r) between every pair of attributes

Be Careful! The correlation coefficient only measures linear correlations and may miss nonlinear relationships entirely.

Create new attribute combinations from existing attributes

Prepare the Data for ML Learning Algorithms

Don’t do this manually, create functions you can reuse.

Data Cleaning

- Fix missing features
    - We did this by filling the gaps with median values.
- Handling text and categorical attributes (pg 65)
    - Most machine learning models prefer numbers, so we need to convert the text / categories to numerical values.
        - OneHotEncoder: Convert a text attribute to a value of 1 and leave the others as 0.
        - We could also create different numerical values for each category.

Custom Transformers

- SciKit provides many useful transformers, but you’re going to want to create your own.
- Needed for:
    - Custom cleaning operations
    - Combining specific attributes
- SciKit Learn relies on duck typing, no inheritance.
    - All you need to do is create a custom class and implement:
        - fit() returning itself
        - transform()
        - fit_transform()

Feature Scaling

- An important transformation we should know how to apply to our data is **feature scaling** to fix the range scales of our data.
- Two common ways:
    - Min-Max scaling (**Normalization**)
        - Values are shifted and rescaled so they range from 0 to 1
    - **Standardization**
        - First subtract the mean and then divide by the standard deviation, the resulting distribution has unit variance

Transformation Pipelines

There are many data transformation steps we need to execute in the right order.

We can use SciKit-Learn’s pipeline class to help create these sequences.

We create a numerical pipeline to handle the numerical attribute columns

We also create a full pipeline to run our transformations through all the columns, combining the numerical pipeline with the OneHotEncoder for the categorical columns.

Select and Train a Model

Training and Evaluating on the Training Set

Let us first try a Linear Regression Model for our housing prices task

We look at the RMSE and see that the model is underfitting the training data.

- Either the features aren't good enough info to make predictions
- Or the model isn't powerful enough
    - Let's try a more powerful model.

Let's try a **Decision Tree Regressor**

- DTR is good at finding complex non-linear relationships in the data
- We look at the RMSE and see the error is 0.0; this doesn't look right.
- We need to use **cross validation** to split the training data into a smaller set and validation set
    - We can use the K-fold cross validation to split the training set into 'k' folds.
- Oh no! Now that we've tried the DTR we can see it is overfitting so much it performs worse than the linear regression model.

Let's try a **Random Forest Regressor**

- Works by training many decision trees on random subsets of the features and averaging out their predictions.
- Building a model on top of many other models is called **Ensemble Learning**
- We see that this RFR model is performing better than the previous two, but not by much.
    - We notice the RMSE on the training set is much lower than on the validation set so we know the model is still overfitting the training set.

At this point, we should try out a few more models from various categories of machine learning algorithms.

Our goal at this point is to shortlist a few promising models.

Fine Tune your Model

Tips on fine-tuning our shortlisted models.

Fiddle with the hyperparameters to evaluate possible combinations.

This can be done with:

- **GridSearchCV**: Pass it the hyperparameters and it will use cross-validation to evaluate all possible combinations
    - Good for relatively few combinations, bad when hyperparameter space is large.
- **RandomizedSearchCV**: Evaluates a given number of random combinations.
- You can also use **Ensemble Methods** to combine models to perform better (Ch7)

Analyze the best models and their errors:

- Inspect the **relative importance** of each attribute for making accurate predictions and drop the less useful features.
- Look at the specific errors your system makes and try to understand them,

Evaluate your system on the Test Set

We can compute a 95% confidence interval for the generalization error.

- We see that our system performs worse than the real estate experts' price estimates, but may be good enough to launch in order to free up more of their time for other tasks.

Launch, Monitor, and Maintain your System

Chapter 3: Classification

Task: To create a classification system on the MNIST dataset of 70,000 handwritten digits.

There are 70,000 images each with 784 pixel intensity (0-255) features representing a 28 x 28 image.

Training a Binary Classifier

Let us start simply by training a binary classifier to identify if a digit is a 5 or not 5.

A good first pick for classification tasks is **Stochastic Gradient Descent (SGD)**

- Good for large datasets and online learning (one training instance at a time)
- In our jupyter notebook we see that it can correctly predict a ‘5’

Performance Measures

Evaluating a classifier is a lot harder than evaluating a regressor, let’s see how

Measuring Accuracy using Cross-Validation

At first glance, using cross-validation to test our classifier looks good, it's showing above 93% accuracy on all cross-validation folds.

Before we get too excited, lets comparing this to a dumb classifier that just classifies every image as ‘not 5’

…

And we see that simply classifying any image as ‘not 5’ has 90% accuracy! This is because only about 10% of our images are 5s.

This shows why accuracy is generally not a good performance measure for classifiers, especially when working with **skewed datasets**

Confusion Matrix

A better way to evaluate the performance of a classifier is to use a confusion matrix.

The confusion matrix is used to count the number of times instances of class A were classified as class B.

Not 5s	53892 correct classifications as non 5s (**true negatives**)	687 incorrect classifications as 5s (**false positives**)
5s	1891 incorrect classifications	3530 correct classifications

	as non 5s (**false negatives**)	as 5s (**true positives**)

A perfect classifier would only have true negatives and true positives.

This gives us a lot of information, but there are more concise metrics.

Let’s look at the accuracy of the positive predictions, i.e. the **precision** of our classifier.

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

This is used alongside **recall** (also called sensitivity or true positive rate) which is the ratio of positive instances that are correctly detected by the classifier.

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

Calculating the precision and recall of our ‘5 classifier’ shows that our precision is 83% and our recall is 65% meaning that our ‘5 classifier’ is correct 83% of the time and only detects 65% of the 5s it sees.

It is convenient to combine Precision and Recall together to find the F1 score which is the harmonic mean of precision and recall. The classifier only gets a high F1 score if both precision and recall are high.

$$F1 = \frac{2}{\frac{1}{precision}+\frac{1}{recall}} = 2 * \frac{precision * recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

The F1 score is a convenient metric to use but not always what you want. Depending on your system you may want higher precision with less emphasis on recall or vice versa. But you can’t always have it both ways - increasing precision reduces recall and vice versa. This is called the **precision / recall trade off** (pg 93/94)

Our SGD Classifier makes a classification decision based on the score from it’s decision function. If the score is higher than the decision function threshold, the instance is classified as positive, if not, negative.

We can change the threshold of the decision function in order to change the precision / recall.

In order to decide on which threshold to use, we turn to cross-validation again. See the jupyter notebook.

The ROC Curve

The **Receiver Operating Characteristic Curve** is another common tool used with binary classifiers similar to the precision/recall curve.

The ROC curve plots the true positive rate (TPR / recall / sensitivity) against the false positive rate (FPR)

The FPR is equal to 1 - true negative rate (TNR / specificity)

Thus, the ROC Curve plots the sensitivity versus 1 - specificity.

See the jupyter notebook for the ROC Curve.

One way to compare classifiers is to measure the **area under the curve (AUC)** of the ROC Curve for different classifiers.

A perfect classifier will have an ROC AUC of 1, and a purely random classifier will have an ROC AUC of 0.5

Which should you use? The Precision/Recall Curve, or the ROC Curve?

- As a rule of thumb:
    - Prefer the PR curve when the positive class is rare or when you care more about false positives than false negatives.
    - Otherwise, use the ROC Curve.

In our case for our ‘5 classifier’ while the ROC AUC score looks good, our PR Curve shows us that there is definitely room for improvement.

Next we’ll try a RandomForestClassifier and compare its ROC curve and ROC AUC Score

From our graph in the jupyter notebook we can see the RandomForestClassifier is superior to the SGDClassifier because the ROC Curve is much closer to the top left corner and has a greater AUC value

Calculating the Precision and Recall of the RandomForestClassifier also shows much better results (precision: 99%, recall: 86%)

Multiclass Classification

Multiclass Classifiers can distinguish between more than two classes.

Some algorithms can immediately be used as multiclass classifiers, but others only work as binary classifiers, but there are strategies to use multiple binary classifiers together to create a multiclass classifier.

One strategy is to train a classifier to detect numbers 0-9, you could train 10 binary classifiers.

- This is called the one-versus-the-rest / one-versus-all (OvR) strategy.

Another strategy is to train a binary classifier for every pair of digits (ex. 0 or 1, 0 or 2…)

- This is called the one-versus-one (OvO) strategy.
- If there are N classes, you need to train N * (N -1) / 2 classifiers
    - For the MNIST problem, N = 10, therefore we would need 45 binary classifiers total!

Some algorithms (such as SVM classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train fewer classifiers on larger training sets.

For most binary classification problems OvR is preferred.

Error Analysis

At this point in the project you would now follow the steps in the ML project checklist (Appendix B) to explore data preparation options, try out different models and shortlist the promising ones, and tuning hyperparameters using GridSearchCV, automating as much as possible.

Assuming we have found a promising model, the next step is to find ways to improve it.

Let’s analyze the types of errors it makes.

Let’s start by looking at the confusion matrix. (See the jupyter notebook)

We can see that the most errors come from many images getting incorrectly classified as 8s.

Also there are some errors that 3s are getting classified as 5s, and 5s getting classified as 3s.

With these insights, we can look for ways to improve our classifier

- We could put our efforts into reducing the false 8s.
    - We could try to gather more training data for digits that look like 8s.
    - We could engineer new features to help the classifier
        - Ex. we could create an algorithm to count the number of closed loops (8 has 2, 6 has one, 5 has none)
    - We could preprocess the images to make patterns (like loops) stand out more.

Let’s look at some of the individual errors to gain insights (See the jupyter notebook)

Looking at the plot in the jupyter notebook, some digits are so badly written that even a human would have trouble classifying them, but most misclassified images are obviously wrong to us and it's hard to understand why the classifier made those mistakes.

In this case, the reason these errors exist is because the SGDClassifier is a simple linear model, all it does is assign a weight per class to each pixel, and when it sees an image all it does is sum up the weighted pixel intensities to get a score for each class.

So since 3s and 5s only differ by a few pixels, this model will easily confuse them.

One possible solution to the 3s 5s confusion is to preprocess the images to ensure they are well centered and not too rotated. This will help reduce other errors as well.

Multilabel Classification

Until now each instance has always been assigned to just one class.

In some cases however, you may want to output multiple classes for each instance.

Such a classification system that outputs multiple binary tags is called a **multilabel classification system**.

For example, let’s create a system that looks for values larger than 6, and odd values.

(See jupyter notebook)

There are many ways to evaluate a multilabel classifier, but the best evaluation strategy depends on your project.

- One way is to compute the F1 score for each individual label and take the average or the weighted average

Multioutput Classification

The last type of classification task we are going to discuss is **multioutput multiclass classification (multioutput classification)**

To demo this we will create a system to remove noise from images.

Input will be a noisy image, and output will be a clean image. (Pixel intensities in, pixel intensities out)

The output will be multilabel (one label per pixel) and each label can have multiple values (in this case a value from 0 to 255)

Chapter 4: Training Models

I’m going to be taking notes directly in the jupyter notebook from here on out as it doesn’t seem to be the best method to keep my notes separated like this in two different documents and having to keep jumping between them.