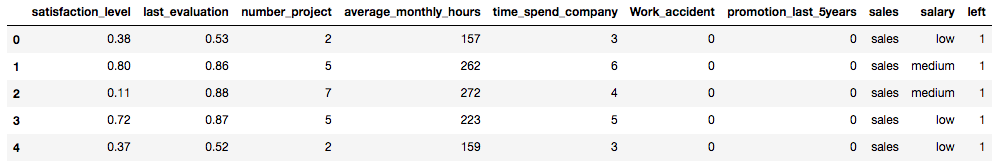
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | **Hope Foundation’s**  **Finolex Academy of Management and Technology, Ratnagiri** | | | | | | | | | |
| **Information Technology Department** | | | | | | | | | |
| Subject name: Cloud Service Design Lab | | | | | | | | Subject Code: ITL603 | | | |
| Class | | TE IT | | Semester – VI (CBCGS) | | | | Academic year: 2018-19 | | | |
| Name of Student | | **Kazi Jawwad A Rahim** | | | | | **QUIZ Score : 10/10** | | | | |
| Roll No | | **27** | | | Assignment/Experiment No. | | | | | 07 | |
| **Title: Case study on Deep Web and Big Data** | | | | | | | | | | | |
|  | | | | | | | | | | | |
| **1.Course objectives applicable**  **COB3**. To understand importance of deep web  **COB6**.To understand the importance of big data | | | | | | | | | | | |
| **2. Course outcomes applicable:**  **CO1** -To define big data and deep web for application development  **CO6**-To Use and Examine different cloud computing services | | | | | | | | | | | |
| **3. Learning Objectives:**   1. To understand concept of big data 2. To understand the concept of deep web | | | | | | | | | | | |
| **4. Practical applications of the assignment/experiment: In cloud environment** | | | | | | | | | | | |
| **5. Prerequisites**:   1. Prior knowledge of big data and deep web 2. Internet Access | | | | | | | | | | | |
| **6. Hardware Requirements**:   1. Internet Access with Browser   **7. Software Requirements:**  Browser like Chrome, Internet Explorer Edge | | | | | | | | | | | |
|  | | | | | | | | | | | |
| **8. Quiz Questions (if any): (Online Exam will be taken separately batchwise, attach the certificate/ Marks obtained)**   1. What is deep web? 2. What is an big data analysis? | | | | | | | | | | | |
|  | | | | | | | | | | | |
| **9. Experiment/Assignment Evaluation:** | | | | | | | | | | | |
| **Sr. No.** | **Parameters** | | | | | | | | **Marks obtained** | | **Out of** |
| **1** | Technical Understanding (Assessment may be done based on Q & A **or** any other relevant method.) Teacher should mention the other method used - | | | | | | | |  | | 6 |
| **2** | Neatness/presentation | | | | | | | |  | | 2 |
| **3** | Punctuality | | | | | | | |  | | 2 |
| **Date of performance (DOP)** | | |  | | | **Total marks obtained** | | |  | | **10** |
| **Date of checking (DOC)** | | |  | | | **Signature of teacher** | | | | | |

**1. Case Study: Binary Classification**

We will be using the Human Resources Analytics [dataset](https://www.kaggle.com/ludobenistant/hr-analytics) on Kaggle. We’re trying to predict whether an employee will leave based on various features such as number of projects they worked on, time spent at the company, last performance review, salary etc. The dataset has around 15,000 rows and 9 columns. The column we’re trying to predict is called “left”. It’s a binary column with 0/1 values. The label 1 means that the employee has left.



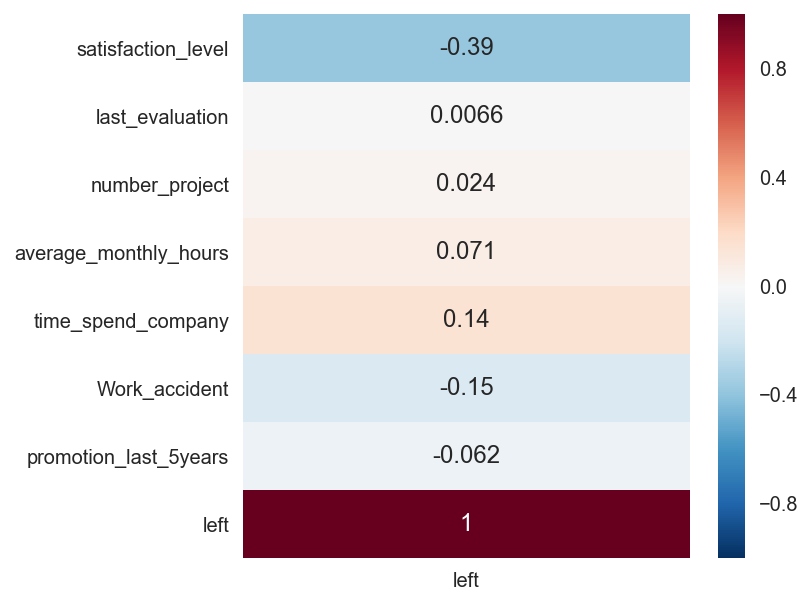
**1.1) Data Visualization & Preprocessing**

First things first, let’s perform some data visualization and preprocessing before jumping straight into building the model. This part is crucial, since we need to know what type of features we are dealing with. For every ML task, we at least need to answer the following questions:

* What type of features do we have: real valued, categorical, or both?
* Do any of the features need normalization?
* Do we have null values?
* What is the label distribution, are the classes imbalanced?
* Is there a correlation between the features?

The jupyter notebook contains the detailed analysis. In summary, there are both real and categorical features. There are no null values, but some features need normalization. 76% percent of the examples are labeled as 0, meaning the employee didn’t leave.

Let’s check the correlation of the features with the labels (the column named “left”). We will use the *seaborn*package for the correlation plot.



In this plot, positive values represent correlation and negative values represent inverse correlation with the label. Of course “left” has perfect correlation with itself, you can ignore that. Other than that only one feature has a strong signal, which is the “satisfaction\_level”, inversely correlated with whether the employee has left. Which makes sense.

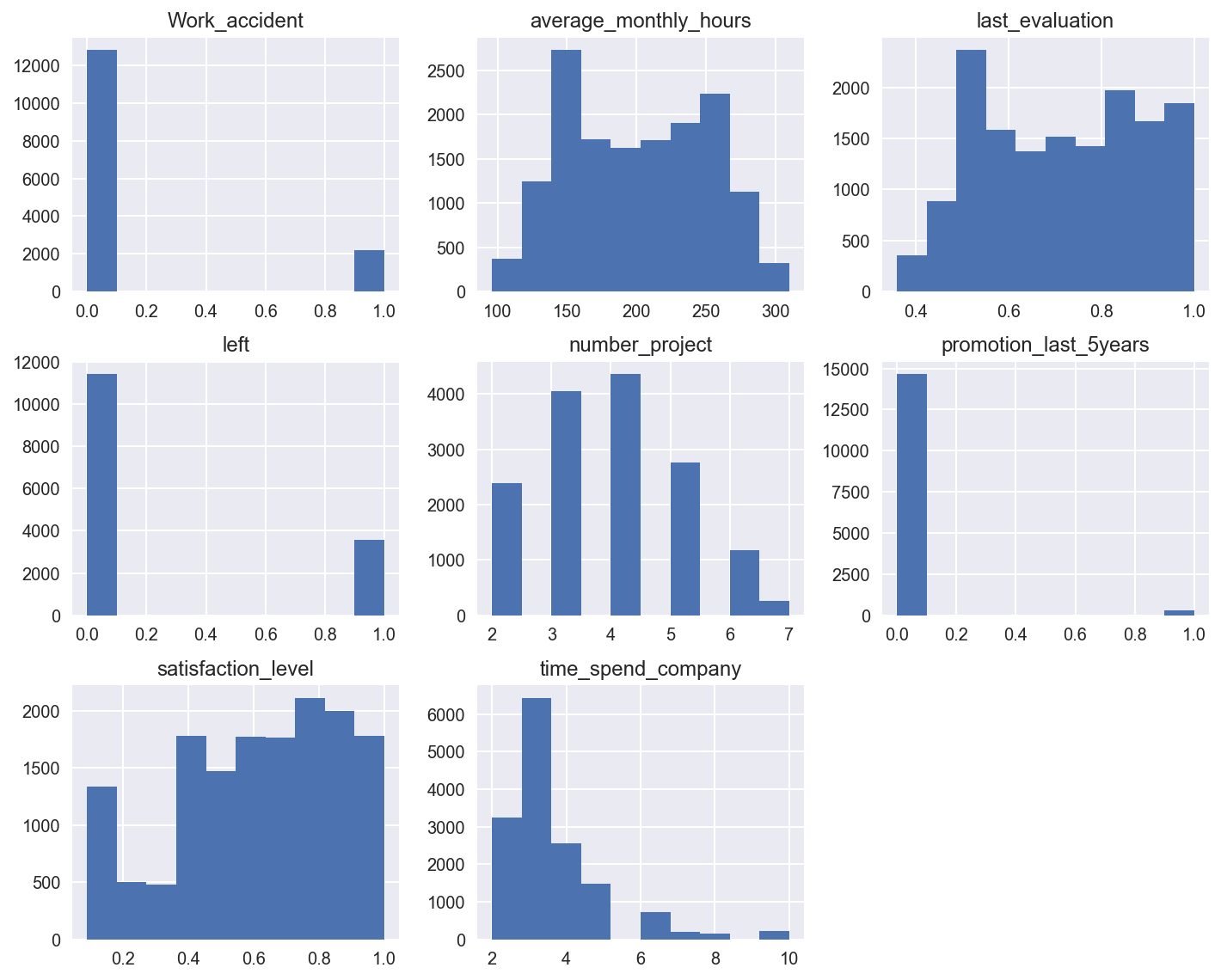
Now let’s look at the pairwise correlation of all features with one another.



We see that “average\_monthly\_hours” is positively correlated with “number\_project”, which again makes sense. The more projects a person is involved with, the more hours of work they need to put in.

Now let’s look at the distribution of feature values. By inspecting the histograms of features we can see which ones need normalization. What’s the motivation behind this? What does normalization mean and why is it needed? Most ML algorithms perform better if the real valued features are scaled to be in a predefined range, for example [0, 1]. This is particularly important for deep neural networks. If the input features consist of large values, deep nets really struggle to learn. The reason is that as the data flows through the layers, with all the multiplications and additions, it gets large very quickly and this negatively affects the optimization process by saturating non-linearities. We will see the detailed demonstration of this in another article, for now we need to pay attention to feature values to be small numbers.

Looking at feature histograms, we need to normalize 3 of the features: average\_monthly\_hours, number\_project, and time\_spend\_company. All other features are within [0, 1] so we can leave them alone.



Scikit-learn has several normalization methods, what we will use is *StandardScaler*. It individually scales the features such that they have zero mean and unit variance, so they all belong to a standard *Normal(0, 1)*distribution. Note that this doesn’t change the ordering of the feature values, it just changes the scale. It’s a simple yet extremely important trick.

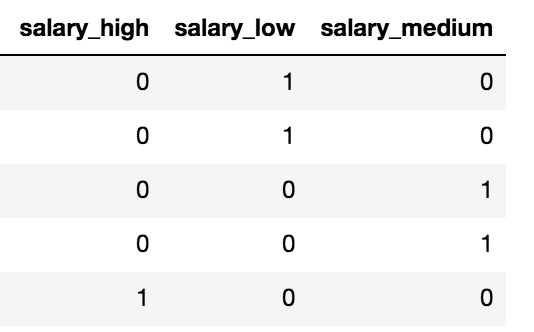
The data we loaded is in a *pandas DataFrame.*Pandas is an extremely popular package to deal with tabular data, especially in an interactive environment like jupyter notebooks. DataFrame is the most commonly used data structure of pandas that acts as a container for our data, and exposes several built-in functions to make our life easier (check out the notebook for more details). In the code snippet below, dfis the DataFrame for our data.

Scikit-learn API is very well designed and contains 4 very commonly used methods. Predictors are ML models like Logistic Regression, and transformers are data manipulators like Standard Scaler.

* *fit*: For predictors performs training on the given input. For transformers computes the statistics like mean and standard deviation of the input to be used later.
* *transform*: For transformers manipulates the input data using the stats learned by the fit function. We run the transform method after fit since there’s a dependency. Predictors don’t support this method.
* *fit\_transform*: Performs fit + transform in a single call efficiently. For transformers, computes the stats of the input and performs the transformation. It’s very a commonly used method with transformers. For predictors, trains the model and performs prediction on the given input.
* *predict:*As its name suggests, for predictors performs the prediction task using the model trained with the fit method. Very commonly used with predictors. Transformers don’t support this method.

Now that we have scaled the real-valued features to be in a desirable range, let’s deal with the categorical features. We need to convert categorical data to *one-hot* representation. For example the salary column contains 3 unique string values: low, medium and high. After one-hot conversion we will have 3 new binary columns: salary\_low, salary\_medium and salary\_high. For a given example, only one of them will have the value 1, the others will be 0. We will then drop the original salary column because we don’t need it anymore.

The one-hot conversion is performed by the *get\_dummies*of pandas. We could have also used the *OneHotEncoder*in scikit-learn, they both get the job done. Since our data is already is in a pandas dataframe, get\_dummies is easier. It also automatically perform the renaming of the features.



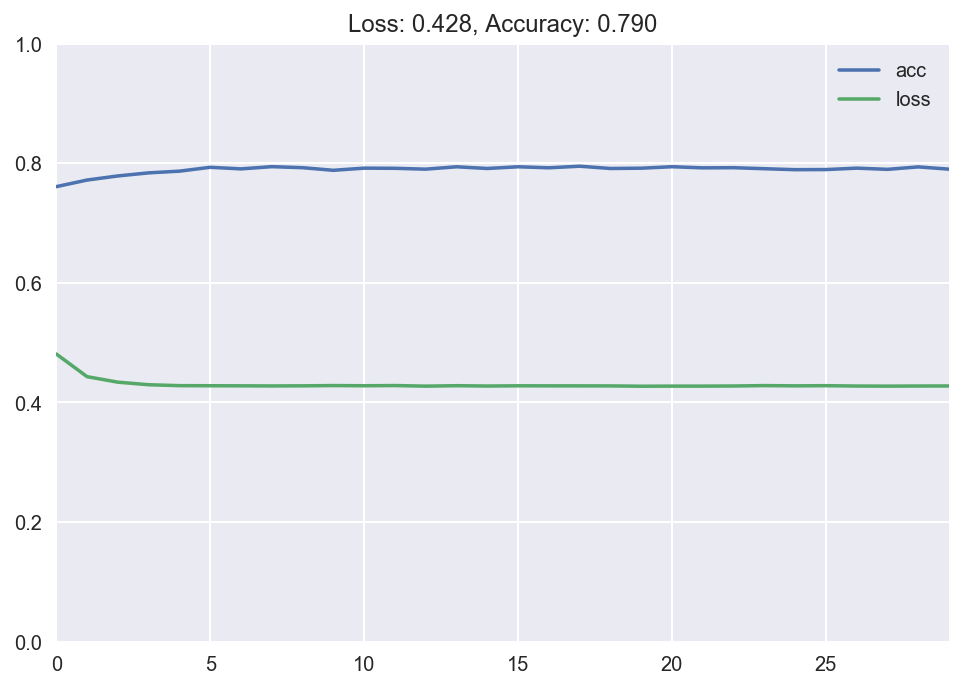
Now comes the final part of creating the training and test data. The model will perform learning on the training set and be evaluated on the held-out test set. Scikit-learn has a convenient *train\_test\_split*function. We only need to specify the fraction of the test set, in our case 30%. But first we convert our data from pandas dataframe to numpy array using the *values* attribute of the dataframe.

**1.2) Logistic Regression Model**

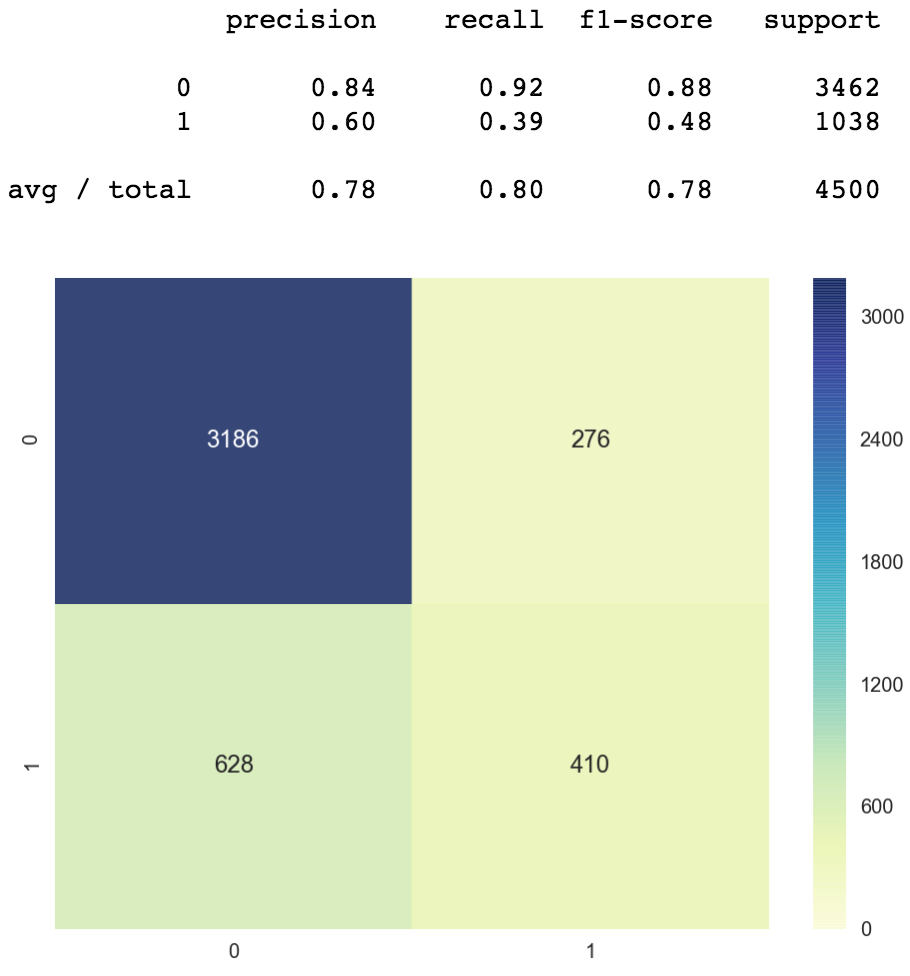
Now that we’re done with the data preprocessing and train/test set generation, here comes the fun part, training the model. We first start with a simple model, Logistic Regression (LR). We will then train a deep ANN and compare the results to LR.

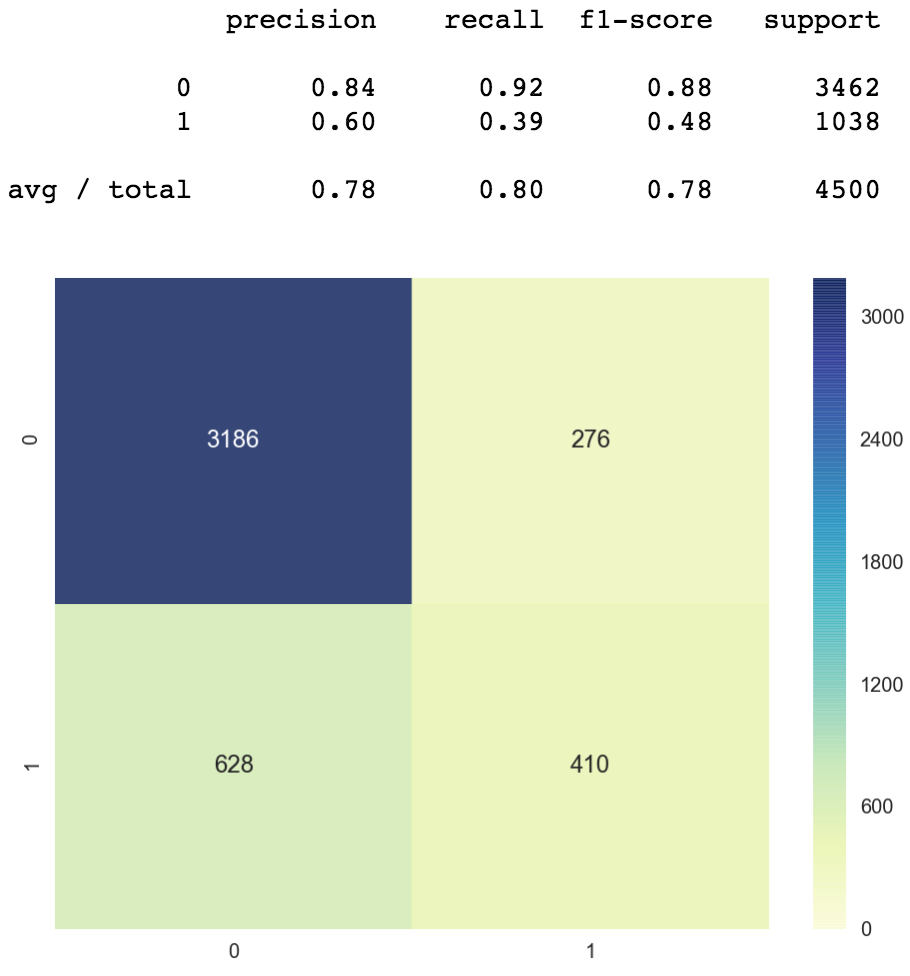
After the first article, building the model should be very familiar.

We get 79% training accuracy. This is actually pretty bad, because above we saw that 76% of the labels were 0. So the most naive classifier which always outputs 0 regardless of the input would get 76% accuracy, and we’re not doing much better than that. This means our data is not linearly separable, just like the examples we saw in the first article, and we need a more complex model.



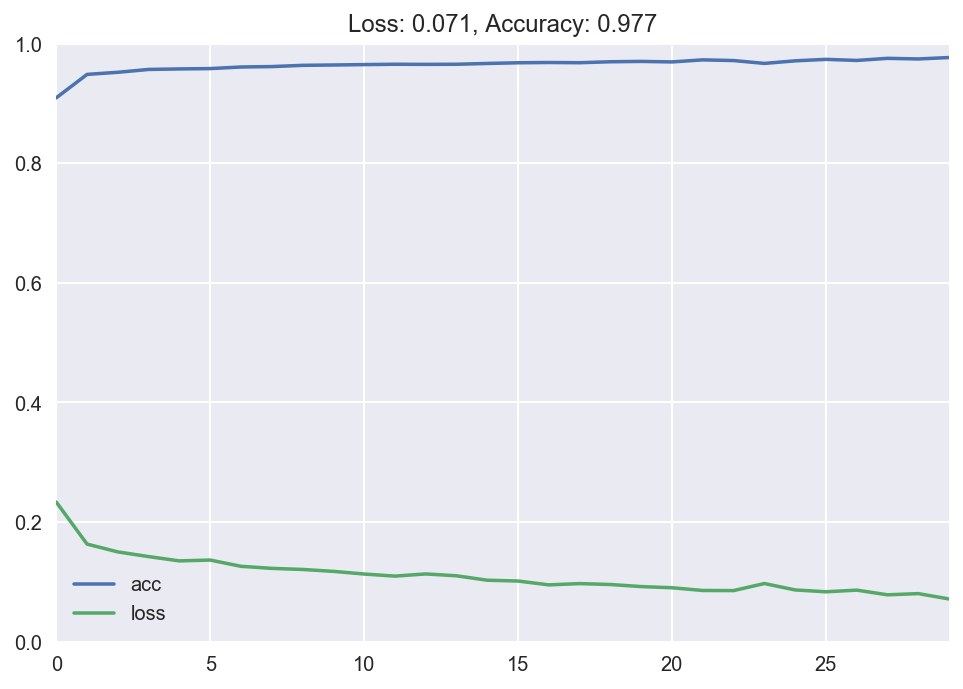
Above chart depicts the training loss and accuracy. But more importantly we’re interested in the metrics of the test set. Metrics in the training set might be misleading since the model is already trained on it, we want to check how the model performs on an held-out test set. Test accuracy is 78%, slightly lower than training accuracy. Test accuracy of ML models are almost always less than training, because the test data is unseen to the model during the training process. Looking at the classification report, we see that only 60% of the examples belonging to class 1 are classified correctly. Pretty bad performance. The confusion matrix also doesn’t look promising showing a lot of misclassified examples.



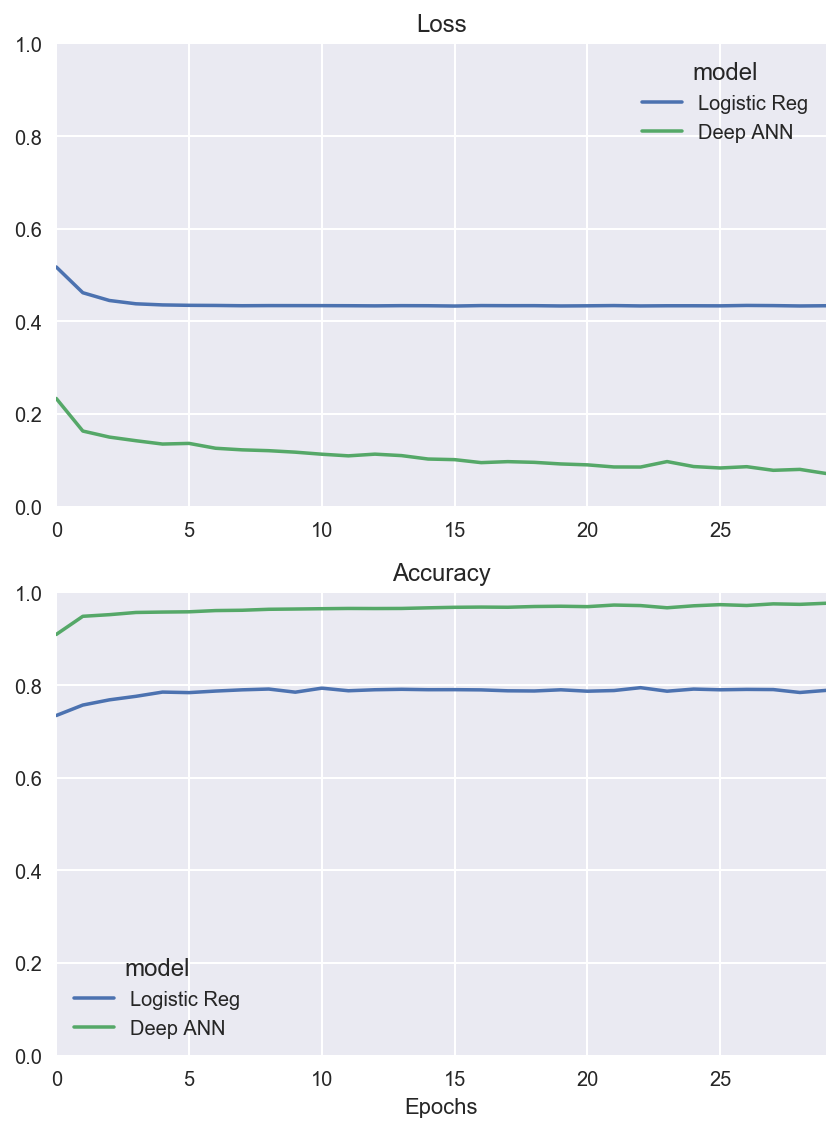


**1.3) ANN Model**

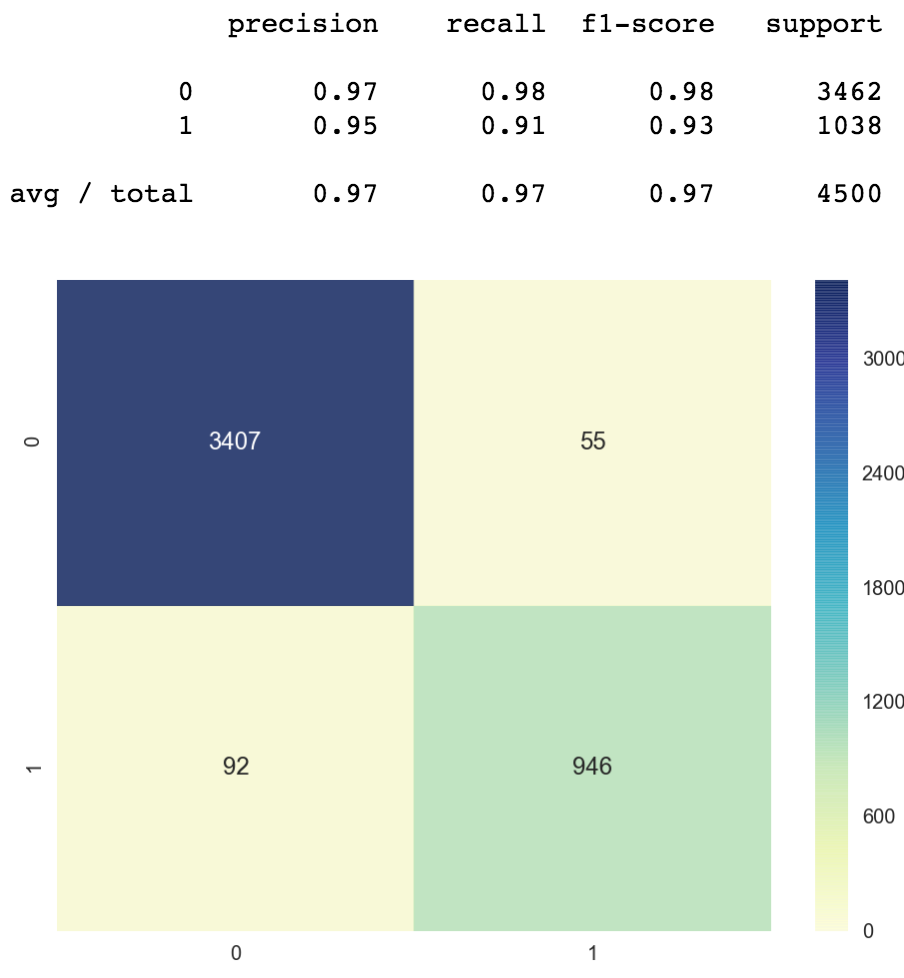
Now let’s build a deep neural network for binary classification. This model will be much more powerful, and will be able to model non-linear relationships. The model building process is again very familiar. We have 2 hidden layers with 64 and 16 nodes with tanh activation function. The output layer uses the sigmoid activation since it’s a binary classification problem. We use the Adam optimizer with learning rate set to 0.01. This time we achieve 97.7% training accuracy, pretty good.



Let’s compare the LR and ANN models. The ANN model is much superior, having a lower loss and a higher accuracy.



And for completeness here’s the classification report and confusion matrix of the ANN model on the test set. We achieve 97% accuracy, compared to 78% of the LR model. We still misclassify 147 examples out of 4500.



We can further improve the performance of the ANN by doing the following:

* Train the model for longer (increase the number of epochs).
* Hyperparamter tuning: change the learning rate, use a different optimizer than Adam (RMSprop for example), use another activation function than tanh (can be relu).
* Increase the number of nodes per layer: Instead of 64–16–1 we can do 128–64–1.
* Increase the number of layers: We can do 128–64–32–16–1.

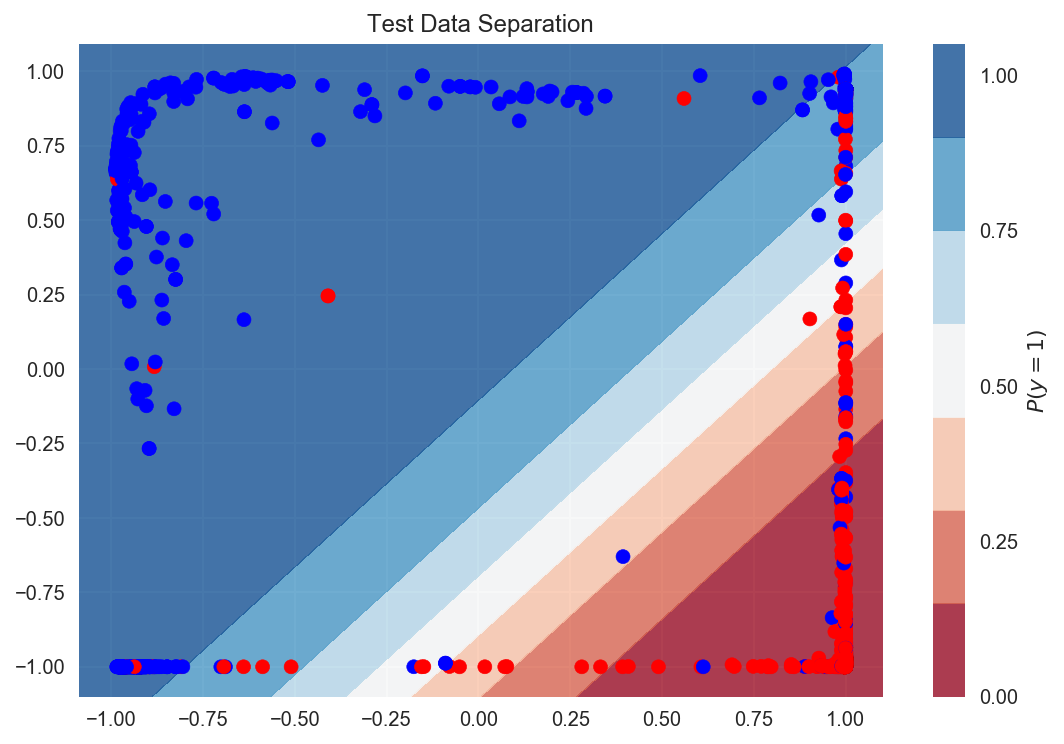
One important caveat though, as we make the model more powerful, the training loss will likely decrease and accuracy will increase. But we will run into the risk of overfitting. Meaning the complex model will perform worse on the test set compared to a simpler model, even though the training metrics of the complex model is better. We will talk more about overfitting in another article, but this is very important to keep in mind. That’s why we don’t go crazy with number of layers and nodes per layer. The simplest model that gets the job done is sufficient.

**1.4) Visualization of Deep ANN**

In the previous article we learned that each layer of the ANN performs a non-linear transformation of the input from one vector space to another. By doing this we are projecting our input data to a new space where the classes are separable from each other via a complex decision boundary.

Let’s visually demonstrate this. Our input data after the initial data preprocessing we did above is 20 dimensional. For visualization purposes let’s project it to 2D. Remember that having k nodes in a layer means that this layer transforms its input such that the output is a k-dimensional vector. The ANN we trained above had two hidden layers with 64 and 16 nodes. Then we need a new layer with 2 nodes in order to project our data to a 2D space. So we add this new layer just before the output node. The rest is completely untouched.

Here’s the resulting projection of our input data from 20D to 2D space. The decision boundary corresponds to the last layer of the ANN. The ANN was able to separate out the classes pretty well, with some misclassifications. A lot of data points overlap in 2D so we can’t see them all, for reference the model misclassifies around 160 points out of 4500 (96% accuracy). We aren’t concerned about accuracy with this model anyway, we are interested in the projection of a high-dimensional input to 2D. This is a neat little trick to visually demonstrate the result of the projections performed by the ANN.



A more principled visualization approach would be using *t-SNE*, which is a dimensionality reduction technique for visualizing high-dimensional data.

**Big Data**

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. The threshold at which organizations enter into the big data realm differs, depending on the capabilities of the users and their tools. For some, it can mean hundreds of gigabytes of data, while for others it means hundreds of terabytes. As tools for working with big data sets advance, so does the meaning of big data. More and more, this term relates to the value you can extract from your data sets through advanced analytics, rather than strictly the size of the data, although in these cases they tend to be quite large. Over the years, the data landscape has changed. What you can do, or are expected to do, with data has changed. The cost of storage has fallen dramatically, while the means by which data is collected keeps growing. Some data arrives at a rapid pace, constantly demanding to be collected and observed. Other data arrives more slowly, but in very large chunks, often in the form of decades of historical data. You might be facing an advanced analytics problem, or one that requires machine learning. These are challenges that big data architectures seek to solve.

Big data solutions typically involve one or more of the following types of workload:

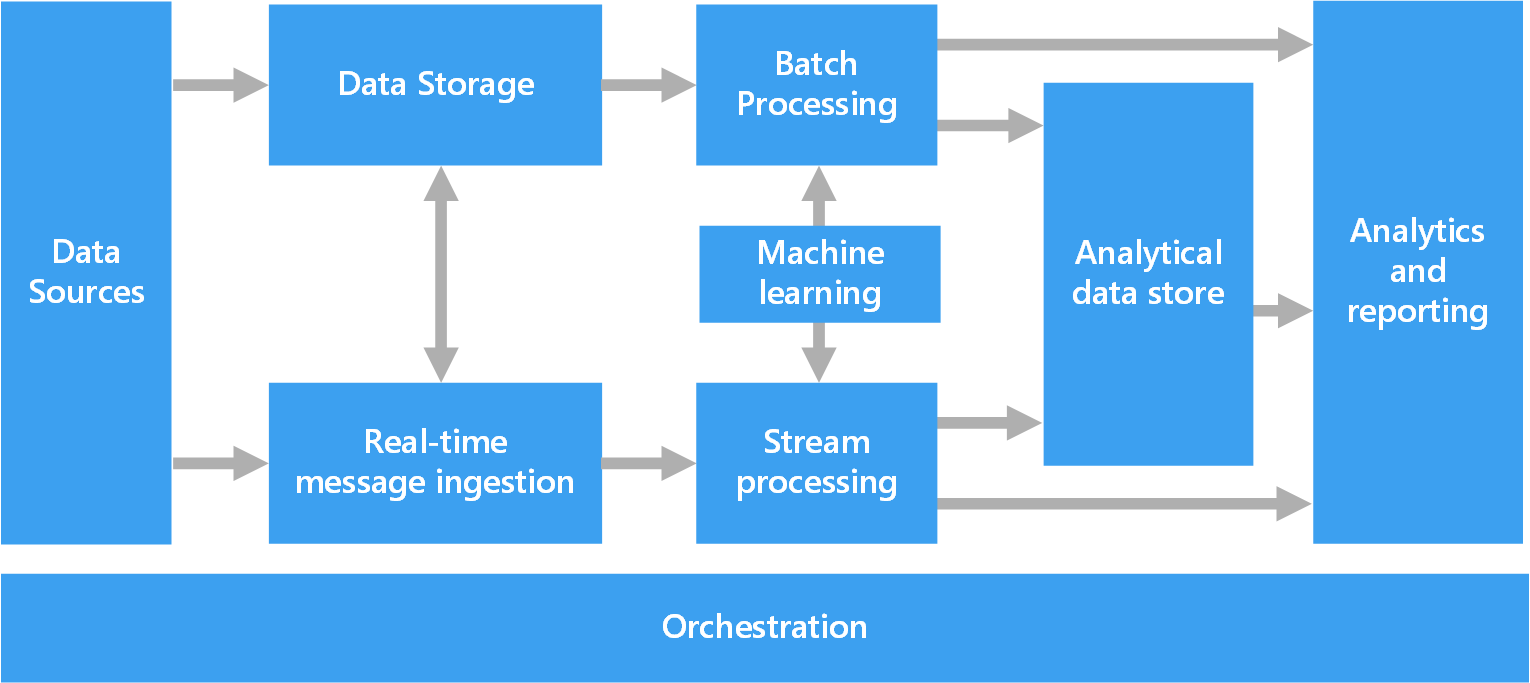
* Batch processing of big data sources at rest.
* Real-time processing of big data in motion.
* Interactive exploration of big data.
* Predictive analytics and machine learning.

Consider big data architectures when you need to:

* Store and process data in volumes too large for a traditional database.
* Transform unstructured data for analysis and reporting.
* Capture, process, and analyze unbounded streams of data in real time, or with low latency.

## Components of a big data architecture

The following diagram shows the logical components that fit into a big data architecture. Individual solutions may not contain every item in this diagram.



Most big data architectures include some or all of the following components:

* **Data sources:** All big data solutions start with one or more data sources. Examples include:
  1. Application data stores, such as relational databases.
  2. Static files produced by applications, such as web server log files.
  3. Real-time data sources, such as IoT devices.
* **Data storage:** Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a data lake. Options for implementing this storage include Azure Data Lake Store or blob containers in Azure Storage.
* **Batch processing:** Because the data sets are so large, often a big data solution must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files, processing them, and writing the output to new files. Options include running U-SQL jobs in Azure Data Lake Analytics, using Hive, Pig, or custom Map/Reduce jobs in an HDInsight Hadoop cluster, or using Java, Scala, or Python programs in an HDInsight Spark cluster.
* **Real-time message ingestion:** If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This might be a simple data store, where incoming messages are dropped into a folder for processing. However, many solutions need a message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics. This portion of a streaming architecture is often referred to as stream buffering. Options include Azure Event Hubs, Azure IoT Hub, and Kafka.
* **Stream processing:** After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis. The processed stream data is then written to an output sink. Azure Stream Analytics provides a managed stream processing service based on perpetually running SQL queries that operate on unbounded streams. You can also use open source Apache streaming technologies like Storm and Spark Streaming in an HDInsight cluster.
* **Analytical data store:** Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. The analytical data store used to serve these queries can be a Kimball-style relational data warehouse, as seen in most traditional business intelligence (BI) solutions. Alternatively, the data could be presented through a low-latency NoSQL technology such as HBase, or an interactive Hive database that provides a metadata abstraction over data files in the distributed data store. Azure SQL Data Warehouse provides a managed service for large-scale, cloud-based data warehousing. HDInsight supports Interactive Hive, HBase, and Spark SQL, which can also be used to serve data for analysis.
* **Analysis and reporting:** The goal of most big data solutions is to provide insights into the data through analysis and reporting. To empower users to analyze the data, the architecture may include a data modeling layer, such as a multidimensional OLAP cube or tabular data model in Azure Analysis Services. It might also support self-service BI, using the modeling and visualization technologies in Microsoft Power BI or Microsoft Excel. Analysis and reporting can also take the form of interactive data exploration by data scientists or data analysts. For these scenarios, many Azure services support analytical notebooks, such as Jupyter, enabling these users to leverage their existing skills with Python or R. For large-scale data exploration, you can use Microsoft R Server, either standalone or with Spark.
* **Orchestration:** Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard. To automate these workflows, you can use an orchestration technology such Azure Data Factory or Apache Oozie and Sqoop.

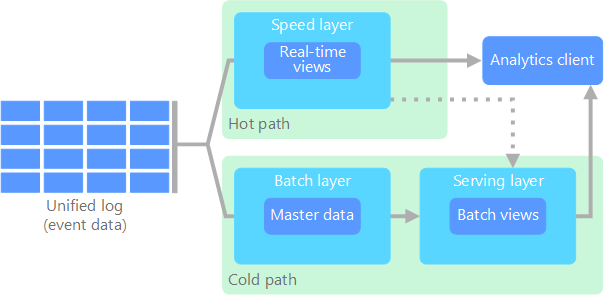
## Lambda architecture

When working with very large data sets, it can take a long time to run the sort of queries that clients need. These queries can't be performed in real time, and often require algorithms such as MapReduce that operate in parallel across the entire data set. The results are then stored separately from the raw data and used for querying. One drawback to this approach is that it introduces latency - if processing takes a few hours, a query may return results that are several hours old. Ideally, you would like to get some results in real time (perhaps with some loss of accuracy), and combine these results with the results from the batch analytics.

The lambda architecture, first proposed by Nathan Marz, addresses this problem by creating two paths for data flow. All data coming into the system goes through these two paths:

* A batch layer (cold path) stores all of the incoming data in its raw form and performs batch processing on the data. The result of this processing is stored as a batch view.
* A speed layer (hot path) analyzes data in real time. This layer is designed for low latency, at the expense of accuracy.

The batch layer feeds into a serving layer that indexes the batch view for efficient querying. The speed layer updates the serving layer with incremental updates based on the most recent data.



Data that flows into the hot path is constrained by latency requirements imposed by the speed layer, so that it can be processed as quickly as possible. Often, this requires a tradeoff of some level of accuracy in favor of data that is ready as quickly as possible. For example, consider an IoT scenario where a large number of temperature sensors are sending telemetry data. The speed layer may be used to process a sliding time window of the incoming data.

Data flowing into the cold path, on the other hand, is not subject to the same low latency requirements. This allows for high accuracy computation across large data sets, which can be very time intensive.

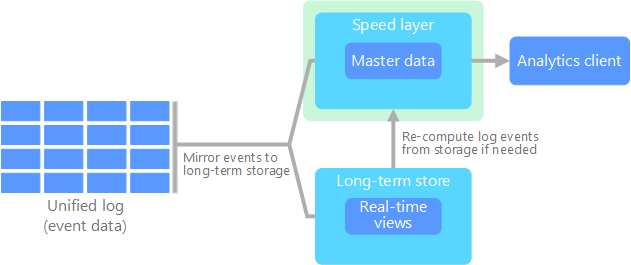
Eventually, the hot and cold paths converge at the analytics client application. If the client needs to display timely, yet potentially less accurate data in real time, it will acquire its result from the hot path. Otherwise, it will select results from the cold path to display less timely but more accurate data. In other words, the hot path has data for a relatively small window of time, after which the results can be updated with more accurate data from the cold path.

The raw data stored at the batch layer is immutable. Incoming data is always appended to the existing data, and the previous data is never overwritten. Any changes to the value of a particular datum are stored as a new timestamped event record. This allows for recomputation at any point in time across the history of the data collected. The ability to recompute the batch view from the original raw data is important, because it allows for new views to be created as the system evolves.

## Kappa architecture

A drawback to the lambda architecture is its complexity. Processing logic appears in two different places — the cold and hot paths — using different frameworks. This leads to duplicate computation logic and the complexity of managing the architecture for both paths.

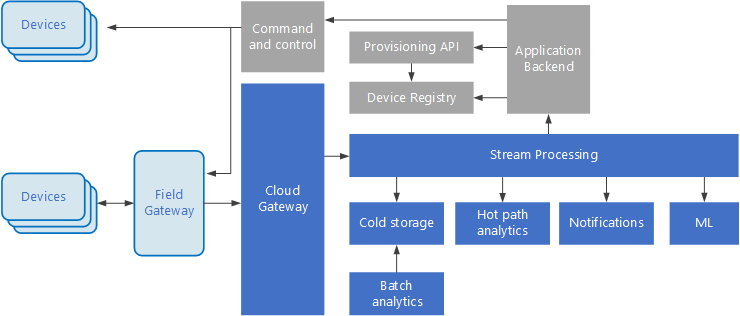
The kappa architecture was proposed by Jay Kreps as an alternative to the lambda architecture. It has the same basic goals as the lambda architecture, but with an important distinction: All data flows through a single path, using a stream processing system.



There are some similarities to the lambda architecture batch layer, in that the event data is immutable and all of it is collected, instead of a subset. The data is ingested as a stream of events into a distributed and fault tolerant unified log. These events are ordered, and the current state of an event is changed only by a new event being appended. Similar to a lambda architecture speed layer, all event processing is performed on the input stream and persisted as a real-time view. If you need to recompute the entire data set (equivalent to what the batch layer does in lambda), you simply replay the stream, typically using parallelism to complete the computation in a timely fashion.

## Internet of Things (IoT)

From a practical viewpoint, Internet of Things (IoT) represents any device that is connected to the Internet. This includes your PC, mobile phone, smart watch, smart thermostat, smart refrigerator, connected automobile, heart monitoring implants, and anything else that connects to the Internet and sends or receives data. The number of connected devices grows every day, as does the amount of data collected from them. Often this data is being collected in highly constrained, sometimes high-latency environments. In other cases, data is sent from low-latency environments by thousands or millions of devices, requiring the ability to rapidly ingest the data and process accordingly. Therefore, proper planning is required to handle these constraints and unique requirements. Event-driven architectures are central to IoT solutions. The following diagram shows a possible logical architecture for IoT. The diagram emphasizes the event-streaming components of the architecture.



The cloud gateway ingests device events at the cloud boundary, using a reliable, low latency messaging system. Devices might send events directly to the cloud gateway, or through a field gateway. A field gateway is a specialized device or software, usually collocated with the devices, that receives events and forwards them to the cloud gateway. The field gateway might also preprocess the raw device events, performing functions such as filtering, aggregation, or protocol transformation.

After ingestion, events go through one or more stream processors that can route the data (for example, to storage) or perform analytics and other processing.

The following are some common types of processing. (This list is certainly not exhaustive.)

* Writing event data to cold storage, for archiving or batch analytics.
* Hot path analytics, analyzing the event stream in (near) real time, to detect anomalies, recognize patterns over rolling time windows, or trigger alerts when a specific condition occurs in the stream.
* Handling special types of non-telemetry messages from devices, such as notifications and alarms.
* Machine learning.

The boxes that are shaded gray show components of an IoT system that are not directly related to event streaming, but are included here for completeness.

* The device registry is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
* The provisioning API is a common external interface for provisioning and registering new devices.
* Some IoT solutions allow command and control messages to be sent to devices.

**11. Learning Outcomes Achieved**

1.To understand the concept of deep web.

2. To understand the concept of deep web.

**12. Conclusion:**

1. **Applications of the studied technique in industry**
   1. Big data analysis in cloud industry.
2. **Engineering Relevance** 
   1. Use of deep web technology in cloud industry
3. **Skills Developed**
   1. Understanding fundamentals of deep web.
   2. Understanding fundamentals of big data.

**References**:

[1] <https://en.wikipedia.org/wiki/Deep_web>

[2] <https://darkwebnews.com/deep-web/>

[3] <https://en.wikipedia.org/wiki/Big_data>

[4] <https://www.tutorialspoint.com/big_data_tutorials.htm>